Opinnäytetyö (AMK)

Tietotekniikan koulutusohjelma

Ohjelmistotekniikka

2011

Mauno Lempiäinen

PELIMOOTTORIN RAKENTAMINEN HTML5 -YMPÄRISTÖSSÄ

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikan koulutusohjelma | Ohjelmistotekniikka

Marraskuu 2011 | 36

Ohjaaja: TkL Jari-Pekka Paalassalo

Mauno Lempiäinen

PELIMOOTTORIN RAKENTAMINEN HTML5 -YMPÄRISTÖSSÄ

Opinnäytetyössä suunniteltiin ja toteutettiin HTML5-alustalle pelimoottorin sekä siinä pyörivän pelin prototyyppi. Työn tarkoituksena oli ottaa selvää HTML5-standardin sisältämän Canvas -elementin soveltuvuudesta web-pelien alustaksi sekä perehtyä JavaScriptin olio-ohjelmoinnin käytäntöihin ja pelimoottorin rakentamiseen.

Rakennettavan pelimoottorin ei ollut tarkoitus tulla työn aikana valmiiksi, vaan ajatuksena oli saada tehtyä karkea runko, jolla pystytään pyörittämään yksinkertaista peliä ja todentamaan, kannattaako jatkokehitys.

Pelimoottori toteutettiin käyttämällä oliopohjaista JavaScriptiä piirtämään Canvas-elementtiin sen 2D Context -rajapinnan kautta. Pelimoottorin äänissä käytettiin HTML5:n esittelemää audio-elementtiä ja sen arkkitehtuurissa on pyritty ottamaan huomioon laajennettavuus ja jatkokehitys prototyypistä tämän työn jälkeen.

Tuloksena syntynyt prototyyppi pelimoottorista ja siinä toimivasta yksinkertaisesta pelistä, todistavat HTML 5:n potentiaalia pelialustana ja tekniikkana johon kannattaa jatkossa perehtyä. Vaikka itse pelimoottori ei päässyt työn aikana sille asetettuihin tavoitteisiin, oli siihen suunniteltu arkkitehtuuri toimiva ja pelimoottorin jatkokehitys näyttää lupaavalta.

ASIASANAT:

HTML5, Canvas, JavaScript, ohjelmistokehitys, WWW-ohjelmointi, peliohjelmointi, pelimoottori

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information Technology | Software technology

November 2011 | 36

Instructor: Jari-Pekka Paalassalo Tech.Lic., M.Sc.

Mauno Lempiäinen

BUILDING A GAME ENGINE ON HTML5 PLATFORM

The subject of this thesis was to design and build a prototype game engine that runs on the relatively new HTML5 platform using its Canvas element. The purpose was to find out whether the HTML5 can be used as a web-based game engine platform, to study JavaScript as object oriented programming language and gain experience in game programming.

The engine was not meant to be finished during this thesis. The main idea was to get a crude framework of it done so we could run a simple prototype game as a proof of concept and to determine whether further development of the engine is worthwhile.

The game engine was built using object oriented JavaScript to draw on the Canvas element through its 2D context interface. Its sound system was implemented using HTML5 audio elements and the engine architecture was designed with expandability and further development in mind.

The result was successful proof of concept that clearly shows that HTML5 and its Canvas element can be used as a worthy platform for building games and it is a recommendable technology to familiarize yourself with. Even though the resulting game engine didn't meet all of the requirements that was expected of it, the underlying architecture proved to be powerful and further development of the engine seems promising.

KEYWORDS:

HTML5, Canvas, JavaScript, software development, www-programming, game programming, game engine

SISÄLTÖ

OHJELMAT

KUVIOT

KUVAT

1 JOHDANTO

Internet tarjoaa laajan ja lähes maksuttoman jakelukanavan siellä toimiville sovelluksille. Www-sivuilla pyörivien multimedian ja erityisesti pelien käytännössä ainoa varteenotettava alusta on pitkään ollut Adoben Flash. HTML5 on horjuttamassa tätä monopoliasemaa tuomalla vaihtoehtoisen ja ehkäpä monipuolisemman vaihtoehdon kaikkien saataville ilmaiseksi. Tällä hetkellä kehitys näyttää menevän siihen suuntaan, että selaimissa pyörivissä sovelluksissa Flashin kehitys ainakin mobiilialustoille aiotaan lopettaa ja panostaa HTML5:n tuomiin korvaaviin menetelmiin [1]. Tästä syystä tuleville HTML5-osaajille on varmasti kysyntää työmarkkinoilla ja HTML5 on alusta johon kannattaa tutustua.

Työn aiheena on rakentaa 2D-pelimoottorin ja siinä pyörivän pelin prototyyppi käyttäen HTML5:n Canvas-elementtiä sekä JavaScriptiä. Tavoitteena on selvittää soveltuuko HTML5 pelinkehitysalustaksi, oppia käyttämään JavaScriptiä monimuotoisena oliopohjaisena kielenä sekä saada kokemusta pelimoottorin suunnittelusta. Tämä työn ymmärtämistä auttaa, että lukijalta löytyy ohjelmistokehityksen pohjatietämystä ja että olio-ohjelmoinnin peruskäsitteet ovat ennestään tuttuja.

Työn alussa tutustutaan JavaScriptin monimuotoiseen tarjontaan oliopohjaisena kielenä sekä käydään läpi lyhyesti mitä HTML5 tarjoaa pelialustana. Tietopohjaan tutustumisen jälkeen tarkastellaan, missä ympäristössä pelimoottoria on kehitetty ja miten sen eri vaiheet ovat edenneet. Ennen itse työn tuloksena syntyneen pelimoottorin ja siinä pyörivän pelin rakenteen läpikäymistä määritellään, mitä toiminnallisuuksia sen on tarkoitus pitää sisällään ja mitä on tiedostetusti rajattu työn ulkopuolelle.

Lopputuloksena on kevyt pelimoottorin runko ja prototyyppipeli. Pelimoottorista puuttuu monia osia, joita oletetaan löytyvän valmiista pelimoottorista ja sen editorityökalut eivät myöskään kuulu tämän työn rajaukseen. Puutteet on otettu arkkitehtuurissa huomioon mahdollistaen sen jatkokehittämisen tämän työn ulkopuolella.

2 JAVASCRIPTIN KÄYTTÖ OLIOPOHJAISENA OHJELMOINTIKIELENÄ

JavaScript on prototyyppipohjainen komentosarjakieli, jonka käyttö keskittyy pääasiassa internetselaimiin. Kielen esitteli ensimmäisen kerran Netscape, joka käytti sitä Netscape Navigator 2.0 -selaimessaan. Kielen saadessa suosiota ja levitessä käytettäväksi myös toisissa selaimissa sekä muissa ympäristöissä syntyi ECMAScript, joka on yhteinen ohjeistus kielen rakenteesta. Ohjeistus määrittää kielen perusrakenteen ohjelmointikielenä, mutta ei puutu tai huomioi kielen toimimista internetselaimien ympäristössä. Selaimissa käytetty JavaScript noudattaa ECMAScriptin asettamaa ohjeistusta, mutta siihen on myös lisätty selainten vaatimia työkaluja www-sivujen muokkausta varten. Koska nämä lisäykset eivät enää kuulu ohjeistukseen, vaan ovat osa kielen tulkitsijaa, niiden olemassaolo ja toimivuus saattaa vaihdella selainten välillä. [2]

2.1 JavaScriptin ominaisuudet

Ohjelmointikielenä JavaScript on syntaksiltaan hyvin löyhä ja anteeksiantavainen. Tämän vuoksi sillä on helppoa tehdä yksinkertaisia asioita ilman tutustumista itse kieleen tai sen sääntöihin. Ajatuksena taustalla on tarjota kokemattomille ohjelmoijille helppo ja vaivaton sisäänpääsy kieleen. JavaScriptin matalan sisäänpääsykynnyksen ja sen saavuttaman suosion myötä monen www-ohjelmoijan ei tarvitse ikinä syventyä kielen tarjoamiin mahdollisuuksiin ja ominaisuuksiin. Siksi jopa kieltä pitkään käyttäneille saattaa tulla yllätyksenä, että JavaScript on itse asiassa olio-ohjelmointikieli monimuotoisilla ominaisuuksilla. [3]

JavaScript on heikosti tyypittyvä ja dynaaminen kieli. Heikosti tyypittyvällä tarkoitetaan sitä, että muuttujille ei erikseen määritellä, minkä tyyppisiä arvoja ne voivat pitää sisällään, ja että muuttujien tyyppi voi vaihtua muuttujan luonnin jälkeen. Kieli osaa itse päätellä muuttujan tyypin sille annettavasta arvosta. Voi-

daan siis ajatella, että muuttujan tyyppi on sidottu muuttujalle annettuun arvoon, ei muuttujan määrittelyyn. Kielen dynaamisuus mahdollistaa koodin muokkauksen ja lisäyksen ajon aikana. Käytännössä tämä tarkoittaa, että koko ohjelmakoodia ei tarvitse olla saatavilla, kun ohjelmaa ajetaan, vaan sitä voidaan muokata ja lisätä ohjelman ajon aikana. Dynaamisuus myös viittaa siihen, että muuttujien tietotyyppiä voidaan muuttaa kesken ohjelman suorituksen [4].

JavaScriptissä on mahdollista myös antaa funktioita parametreina toisille funktioille ja asettaa funktioita muuttujiin. Tämä mahdollistaa JavaScriptin monipuolisimpiin ominaisuuksiin kuuluvat lambda-funktiot. Lambda-funktiot ovat nimettömiä funktioita, joita ei ole sidottu mihinkään muuttujaan, vaan ovat periaatteessa kertakäyttöisiä. Sen takia ne soveltuvat hyvin parametreina annettavaksi. [2]

Muuttujien määrittelyalue seuraa leksikaalista (eng. lexical) formaattia. Käytännössä tämä tarkoittaa sitä, että globaalit muuttujat näkyvät kaikille olioille. Funktioilla voi olla omia muuttujia, jotka elävät vain niiden sisällä ja ne näkyvät taas kaikille funktion rajojen sisällä oleville tai kutsutuille olioille. [4]

2.2 JavaScriptin hyvät ja huonot puolet

Hyvät puolet

Löyhän syntaksin ja dynaamisuuden hyvinä puolina voidaan pitää alhaisen aloituskynnyksen antamista monelle www-suunnittelijalle, jotka tarvitsevat sivuilleen vain yksinkertaisia toimintoja. Tällöin www-suunnittelijan ei tarvitse olla ohjelmoija tai syventyä kieleen sen enempää, jotta hän pystyy saamaan aikaiseksi haluamansa. Kielen dynaamisuus antaa myös paljon pelivaraa ohjelmoijalle. Ohjelmaan voi halutessaan vaikka hetkellisesti lisätä ominaisuuksia olemassa oleviin olioihin, tehden niihin muutoksia kesken ohjelman suorituksen. Myös lambda-funktiot ovat erittäin käteviä ja mahdollistavat korkeatasoisen funktionaalisen ohjelmoinnin. [2]

Huonot puolet

Sen lisäksi että JavaScriptissä on paljon hyvää, löytyy siitä myös huonoja puolia, jopa sen hyviksi todetuista ominaisuuksista. Vaikka löyhä syntaksi antaa paljon liikkumavaraa kehittäjälle, on siinä myös yksi vakava heikkous: virheiden löytäminen [4]. Koska JavaScript antaa hyvin paljon anteeksi käyttäjälle on välillä erittäin vaikeaa löytää, missä kohdassa sovelluksessa ilmentyvä virhe on. Esimerkiksi muuttujat eivät noudata tiettyä tyyppiä ja funktiot eivät tarvitse ennalta määritettyjä parametrejaan. Monesti sovellus saattaa toimia hyvin, vaikka siinä on selvä virhe, joka tavallisesti löytyisi kielen käännösvaiheessa. Nämä virheet saattavat tulla esille erikoisissa tapauksissa tai aiheuttaa virheitä muualla ohjelmassa. Tämän takia kehittäjän täytyy varmistaa selvästi, miten hän haluaa ohjelmansa toimivan ja mitkä toiminnot ovat sallittavia. Koska ohjelma saattaa toimia hyvin ilman suurempia tarkistuksia, on monella kehittäjällä houkutus jättää muuttujien arvojen tyyppitarkastukset tekemättä, mikä taas tuottaa ongelmia suurissa usean henkilön yhteisissä projekteissa.

Myös yhtenä ongelmana isoissa projekteissa on muuttujien nimiavaruus. JavaScriptissä jokainen muuttuja on julkinen. Tämä tarkoittaa sitä, että niihin pääsee käsiksi mistä tahansa kohtaa ohjelmassa. Jos muuttuja alustetaan olioiden ulkopuolella, ne ovat globaaleja muuttujia ja näkyvät kaikille funktioille ja olioille. On ilmiselvää, että tämä muodostuu helposti ongelmaksi suurissa projekteissa, jotka saatetaan koostaa monista eri kirjastoista joissa voi olla paljon päällekkäisiä muuttujien nimiä.

2.3 JavaScripti olio-ohjelmointikielenä

Kun perinteisessä olio-ohjelmoinnissa puhutaan olioista, tarkoitetaan luokista johdettuja ilmentymiä [2]. Luokissa määritellään olioiden rakenne eli se mitä metodeja ja attribuutteja olio pitää sisällään. Jokainen samasta luokasta johdettu olio sisältää samat muuttujat ja metodit, ainoastaan muuttujien arvot voivat erota eri olioiden välillä. Kun luokka on rakennettu, ei sen määrittelyyn voi enää

vaikuttaa sen ulkopuolelta. Ainoastaan muokkaamalla luokan määrittelyä, voidaan myös muuttaa luokasta johdettujen olioiden toimintaa.

Teknisesti JavaScript ei ole puhdas olio-ohjelmointikieli, vaan se käyttää prototyyppipohjaista olio-ohjelmointiparadigmaa. Prototyyppipohjainen ohjelmointi on olio-ohjelmoinnin tyylisuunta, jossa luokkia ei ole olemassa. Tyylisuuntaa kutsutaan myös luokattomaksi ohjelmoinniksi. Perinteiset olio-ohjelmoinnin käsitteet, kuten perintä, hoidetaan itse olioiden kautta luokasta perimisen sijaan. [6]

2.3.1 Oliot ja niiden rakenne

JavaScriptissä on muutamia yksinkertaisia tietotyyppejä. Niitä ovat numerot, merkkijonot, *boolean* sekä *null* ja *undefined* [2]. Kaikki muut tyypit ovat olioita, jopa funktiot. Olioita voidaan pitää JavaScriptissä nimi-arvo-parien kokoelmina, tietynlaisena assosiatiivisena taulukkona, jossa nimet muodostuvat merkkijonoista ja arvot voivat olla käytännössä mitä tahansa yksinkertaisista tietotyypeistä vaikka toisiin olioihin. Ainoat rajoitukset ovat, että nimen täytyy olla merkkijono eikä arvo voi olla tyyppiä *undefined* [2].

Olioiden rakennetta ei myöskään sidota sen luomishetkellä. Olion sisältämiä attribuutteja pystytään muokkaamaan, poistamaan sekä lisäämään koska tahansa ja mistä tahansa. Attribuuttien arvoihin pääsee käsiksi joko pisteoperaattorilla tai käyttämällä hakasulkuja, kuten esitetään ohjelma 1:ssä

Ohjelma 1 Olioiden attribuutteihin voidaan viitata joko pisteoperaattorilla tai hakasuluilla.

```
/**
Alla luodaan uusi olio jolla on attribuutti attr ja sillä arvo ”attribuutti”
*/
var olio = { attr: ”attribuutti” };
olio.attr;    // pisteoperaattori
olio[”attr”]; // hakasulkeet
```

Ohjelma 1 (jatkuu)

```
//Allaolevaan ei pääse käsiksi pisteoperaattorilla
olio[”normista poikkeava”] = ”attribuutti”;
```

Ainoastaan jälkimmäisellä keinolla pääsee käsiksi attribuutteihin, joiden nimet eivät noudata muuttujien nimeämiskäytäntöä. Jos oliolle antaa arvon attribuutille mitä ei ole olemassa, niin oliolle luodaan attribuutti. Myös tärkeä ominaisuus mitä ei kannata unohtaa on se, että kaikki oliot annetaan referensseinä, ei kopioina. [3]

2.3.2 Prototyypit

Prototyyppipohjaisena olio-ohjelmointikielenä, JavaScriptissä jokainen olio perustuu prototyyppiin. Kun luodaan uusi olio, voidaan valita mikä jo olemassa oleva olio toimii sen prototyyppinä. Käytännössä tämä tarkoittaa sitä, että jos luodaan uusi olio tyyppiä *Object*, joka on yksi sisäänrakennetuista olioista, linkitetään uusi olio *Object*-olion prototyyppiin ja sillä on käytössä *Object*-olion prototyypissä määritellyt metodit sekä attribuutit. Näitä linkkejä voi teoriassa ketjuttaa rajattomasti. Esimerkiksi perinteinen taulukko on olio jolla on oma prototyyppi, mutta se myös perustuu *Object*-olion prototyyppiin. Uusi taulukko linkitetään siis ensin taulukko-olion prototyyppiin, joka on taas linkitetty *Object*-olion prototyyppiin. [5]

Linkki olion prototyypin ja siitä johdetun uuden olion kesken on yksisuuntainen. Se toimii siten, että jos haetaan oliolta jotain attribuuttia tai metodia, niin ensin tarkistetaan löytyvätkö ne luodusta oliosta. Jos ne löytyvät, käytetään suoraan niitä. Jos taas etsittyä ominaisuutta ei löydy, tarkastetaan löytyykö se olion prototyypistä. Mikäli sitä ei löydy olion prototyypistä, tarkistetaan löytyykö ominaisuus siitä prototyypistä, mihin olio perustuu. Tätä kaavaa toistetaan niin pitkään kunnes ominaisuus löytyy tai päädytään prototyyppiin, jota ei enää ole johdettu

mistään, kuten käy ilmi Ohjelmasta 2. Tällöin palautetaan *undefined* tyyppinen arvo. [4]

Ohjelma 2 Prototyyppien kautta rikastettavien olioiden käyttäytyminen sekä niiden ominaisuuksien periytyminen.

```
function Olio(){};     // luodaan uusi tyhjä funktio olio
function Olio2(){};    // Luodaan toinen tyhjä funktio olio
Olio2.prototype = new Olio; // Johdetaan Olio2 Oliosta

// Lisätään Olio:n prototyyppiin uusi metodi
Olio.prototype.metodi = function(){alert(”metodi”)};

// Lisätään Olio2:n prototyyppiin eri metodi
Olio2.prototype.metodi2 = function(){alert(”metodi2”)};
var instanssi = new Olio();   // Luodaan Olio oliosta instanssi
var instanssi2 = new Olio2(); // Luodaan Olio2 oliosta instanssi

// Palauttaa käyttäjälle ponnahdusikkunassa merkkijonon ”metodi”
// Tämä metodi on johdetusta Olio:n prototyypistä
instanssi2.metodi();

// Lisätään instanssi2: een metodi funktio, joka
// korvaa instanssi2:ssa Olio:n prototyypistä perityn metodin
instanssi2.metodi = function(){alert(”korvattu”)};

// Palauttaa käyttäjälle ponnahdusikkunassa merkkijonon ”korvattu”
instanssi2.metodi();

// Palauttaa käyttäjälle ponnahdusikkunassa merkkijonon ”metodi2”
instanssi2.metodi2();
```

Ohjelma 2 (jatkuu)

```
// Palauttaa käyttäjälle ponnahdusikkunassa merkkijonon ”metodi”

instanssi.metodi();


// Alla oleva kutsu palauttaa virheen, koska instanssin prototyypillä
// ei ole metodi2:sta määriteltynä, vaan se löytyy Olio2:sta.

instanssi.metodi2();
```

Ohjelma 2:sta on helppo päätellä, että jos olion prototyyppiin lisätään tai poistetaan toimintoja kaikki oliot, jotka perustuvat kyseiseen prototyyppiin, toimivat välittömästi niiden muutosten mukaisesti, mikäli niillä itsellään ei ole jo entuudestaan samannimisiä toimintoja. Mutta jos itse olioon tehdään muutoksia, muutokset jäävät pelkästään siihen olioon eivätkä vaikuta prototyyppiin josta olio on johdettu.

2.3.3 Funktioiden rooli

Funktiolla on erityinen rooli JavaScriptissä. Ne toimivat kuten funktioiden oletettaisiin toimivan, uudelleenkäytettävinä kokoelmina koodia, mutta ne ovat myös itsessään olioita yhdellä suurella erolla. Sen lisäksi, että ne ovat linkitettyjä *Function.prototype* -prototyyppiin, sisältävät ne myös itse oman prototyyppi attribuutin. Funktion oma prototyyppi on olio jolla on *constructor*-niminen attribuutti jonka arvo on itse funktio. Tämä tekee funktion roolista erikoisen. Funktion oman prototyypin *constructor*-attribuutti suoritetaan aina kun funktiosta luodaan uusi olio. Koska *constructor*-attribuutti on itse funktion runko, niin toimii funktion sisältö myös rakentajafunktiona. Tämä tekee funktion toimintaperiaatteesta hyvin samantapaisen mitä olio-ohjelmoinnissa odotettaisiin luokalta. [2]

Tämän lisäksi funktio eroaa normaaleista olioista siten, että se saa kaksi piilotettua attribuuttia: *this* ja *arguments*. *Arguments*-attribuutti sisältää kaikki funktiolle kutsussa annetut parametrit. Tämä on tärkeää tietää siksi, koska funktiolle voidaan antaa enemmän tai vähemmän parametreja, kuin mitä on asetettu

funktion määrittelyssä ilman, että siitä seuraisi virhettä. *This*-parametri on erittäin tärkeä olio-ohjelmoinnin näkökulmasta, mutta myös hieman ongelmallisesti toteutettu JavaScriptissä, sillä sen arvo riippuu tavasta millä funktiota kutsutaan. [5]

Funktiota voidaan kutsua käytännössä neljällä eri tavalla. Ensimmäisessä tapauksessa funktio ei kuulu millekään oliolle, joten se käyttäytyy kuten perinteinen funktio. Tällöin jos funktion sisällä käytetään sen *this*-attribuuttia, viittaa se globaaliin objektiin. Tämän vuoksi sen myöskään sisäiset funktiot eivät pääse käsiksi ulkoiseen funktioon. [5]

Toisessa tapauksessa funktiota kutsutaan *new*-avainsanalla ja siitä luodaan uusi olio. Tällöin funktio käyttäytyy kuten olio-ohjelmoinnin luokasta johdettu ilmentymä ja *this*-attribuutti viittaa luotuun olioon. Uusi olio pääsee *this*-attribuutin avulla käsiksi omiin yksilöllisiin arvoihinsa. Luotu olio linkitetään samalla funktion prototyyppiin. Tällaisten funktioiden yleisenä nimeämiskäytäntönä on pidetty nimen alkukirjaimen kirjoittamista suurella, jotta tiedetään tämän funktion omaavan rakentajafunktion. [5]

Kolmas tapa on kutsua funktiota olion metodina. Tällöin kutsutun funktion sisällä *this*-attribuutti viittaa olioon, josta funktiota kutsutaan, eikä itse funktion sisällä oleviin arvoihin. [5]

Neljäs, sekä viimeinen tapa liittyy *Function*-olion metodiin *apply*, jonka prototyypin kaikki funktiot perivät. Metodia voidaan kutsua millä tahansa funktiolla ja se ottaa ensimmäisenä parametrina olion, josta tulee kutsuttava funktion *this*-attribuutti. Toisena argumenttina se ottaa vastaan taulukon parametreja. [5]

2.3.4 Perintä

Perintä on olio-ohjelmoinnin yksi keskeisimmistä ajatuksista. Prototyyppipohjaisilla kielillä perintä hoidetaan kopioimalla olion prototyyppi ja lisäämällä siihen toiminnallisuuksia. JavaScriptissä periaate on sama, mutta se ei noudata sitä aivan täysin, vaan lainaa hieman klassisesta olio-ohjelmoinnista käyttämällä

funktioiden runkoa rakentaja metodina. Perintää voidaan käytännössä hoitaa monella eri tapaa. Jotkut suosivat *Object*-olion rikastuttamista uusilla metodeilla, jotka tekevät perinnästä selkeämpää ja lähempänä klassista olio-ohjelmointia. [5]

Työssä päädyttiin käyttämään funktioita prototyypin runkona ja antamalla sille oman käynnistys- ja tuhoamisfunktionsa, joiden nimi muodostu sanoista *init* ja *destruct* liitettynä funktion nimen eteen kuten selviää Ohjelma 3:sta.

Ohjelma 3 Tapa jolla JavaScriptin perintä hoidetaan työssä. Ensin luodaan olio Parent josta peritään olio Child.

```
/* Funktio josta peritään */
function Parent() {
   // Funktion omat muuttujat
   this.parentAttr1 = null;
   this.parentAttr2 = null;
   // Alustajafunktio, jolla asetetaan olion alkuasetelma
   this.initParent = function(param1, param2) {
      this.parentAttr1 = param1;
      this.parentAttr2 = param2;
   }
   this.destructParent = function() {
      // tässä hoidettaisiin resurssien vapautus
   }
}

/* Funktio joka perii Parent-olion */
function Child() {
   // Ensin määritellään sen omat muuttujat, mitkä poikkeavat
   // Parent-oliosta
   this.childAttr1 = null;
   // sen jälkeen kutsutaan Child-olion omaa alustus funktiota
```

Ohjelma 3 (jatkuu)

```
this.initChild = function(param1, param2, param3){
      // Jonka sisällä kutsutaan sen parentin rakentajaa, jos halutaan
    // se suorittaa.
      this.initParent(param1, param2);
      this.childAttr1 = param3;
   }
   // Samalla periaatteella Child-olion tuhoamisfunktiossa voidaan
   // kutsua sen olion tuhoamisfunktiota, mistä Child olio on johdettu
   this.destructChild = function() {
      this.destructParent(); // Parent olion resurrssien vapautus
      // olion omien resurssien vapautus
   }
}
// Lopuksi määritellään, että Child olion prototyyppi on johdettu
// Parent -oliosta
Child.prototype = new Parent;
```

Kuten aikaisemmin mainittiin, on ohjelma 3:ssa esitetty tapa vain yksi monista mahdollisista keinoista hoitaa perintä. Tämä tapa valittiin siksi, että se tuntuu tutummalta, jos on tottunut perinteiseen olio-ohjelmointiin.

2.4 Moniajo

Yksi JavaScriptin rajoittavista puolista on aidon moniajon puute. Kevyissä www-sovelluksissa tämä ei yleensä muodostu ongelmaksi ja sen puutetta korvataan JavaScriptin asynkronisilla toiminnoilla. Asynkroniset toiminnot eivät suoriudu samaan aikaan muun ohjelman kanssa, vaan niitä voidaan suorittaa riippumatta muun sovelluksen tilasta. Yksi esimerkki asynkronisesta funktiosta on *setInter-*

val joka kutsuu sille annettua ohjelman osaa aina tietyn aikavälin jälkeen, kunnes sen käsketään lopettaa.

Tämä voi vaikuttaa moniajon kaltaiselta, mutta siinä on yksi vakava puute. Sen sijaan, että asynkroninen kutsu ajettaisiin erillään muusta sovelluksesta omassa säikeessään, suoritetaan se ohjelman muiden toimintojen luovutettua kontrolli. Käytännössä tämä tarkoittaa sitä, että se ei luovuta suoritusaikaa sovellukselle ennen kuin se on itse suorittanut oman osansa loppuun.

HTML5 tuo mukanaan ratkaisun jolla päästään askeleen lähemmäs aitoa moniajoa. Se esittelee Web Workers-rajapinnan jonka avulla voidaan ajaa taustalla muusta sovelluksesta riippumattomia toimintoja. Web Workers-rajapinnan kautta kutsutut ohjelmat pyörivät omissa säikeissään ja näin eivät tuki tai vie suoritusaikaa niitä kutsuvalta sovellukselta. [7]

Web Workes-rajapinnan huonona puolena on se, että sen kautta suoritetut ohjelmat eivät ole yhteydessä itse sovelluksen joka niitä kutsuu. Ne eivät pysty suoraan muokkaamaan tai näkemään sovelluksen muuttujia ja funktioita, vaan keskustelevat pääsovelluksen kanssa JavaScriptin *event*-mallin kautta lähettämällä viestejä. Tämän takia pääasiallinen käyttö Web Workers -ohjelmille ovat raskasta laskentaa vaativat tehtävät, joiden ei tarvitse suoraan muuttaa ohjelman tilaa vaan palauttaa tuloksia.[7]

3 HTML5 PELIOHJELMOINNIN ALUSTANA

HTML5 on uusin iteraatio www-sivustojen rakenteen määrittelevästä kielestä. Tällä hetkellä HTML:n viides versio on vielä kesken ja sen määritelmää ei ole viimeistelty. Tästä huolimatta monet modernit selaimet tukevat jo siitä löytyviä ja todennäköisesti lopulliseen määrittelyyn päätyviä ominaisuuksia. [8]

3.1 Uudet ominaisuudet

Uusi versio esittelee ison joukon uusia elementtejä ja rajapintoja, jotka tuovat pitkään toivottuja toiminnallisuuksia www-sivuille. Tässä työssä kuitenkin käytetään hyödyksi pientä osaa niistä, joten ei ole tarpeellista kertoa niistä kaikista. Tässä työssä keskitytään pääasiassa uuteen Canvas-elementtiin ja käytetään kevyesti hyväksi audio-elementtiä.

3.2 Canvas-elementti

Pitkään www-sivuilla tapahtuneet animaatiot tai monipuoliset toiminnot ovat vaatineet kolmannen osapuolen tekemiä ohjelmalisäkkeitä asennettuna selaimeen. Näistä ehkä suosituin on ollut Adoben Flash. Tämä on ollut rajoittava asia, sillä monen sivuston toiminta on ollut riippuvainen ohjelmalisäkkeiden tuesta käyttäjän selaimessa ja niiden toimivuudesta. HTML5:n esittelemä Canvas-elementti pyrkii tarjoamaan varteenotettavan vaihtoehdon ohjelmalisäkkeille.

JavaScriptillä on aikaisemmin pystynyt tekemään yksinkertaisia animaatioita, siirtämään elementtejä tai kuvia DOM-puussa sekä muuttamaan niiden tyylejä. Mutta koska DOM-rakennetta ei ole suunniteltu animointialustaksi, ei se taivu kovinkaan monimuotoisiin asioihin. Tähän tarkoitukseen on tehty uudessa HTML-standardissa mukana oleva Canvas-elementti. Se mahdollistaa JavaScriptillä suoraan piirtämisen selaimeen Canvas-elementissä määritellylle alueelle,

joka on käytännössä bittikartta. Itse piirtäminen Canvakselle hoidetaan sen Contextien kautta. [8]

3.3 Canvaksen context-rajapinta

Contextit ovat ohjelmointirajapintoja, joiden avulla piirretään canvakselle. Itse Canvas-elementtiä voidaan miettiä kehyksenä, joka määrittää piirrettävän alueen koon, sijainnin ja tilan. Eri Contextit hoitavat sen mitä ja miten alueelle piirretään. HTML5:n Canvakselle kuuluu oma 2D Context -rajapinta, jota tuetaan yleisempien selainten uusissa versioissa. Sitä myös käytetään tämän työn pelimoottorissa. Huonona puolena siinä on, että sen rajapinta on hyvin suppea ja ei sinällään sovi parhaiten käytettäväksi pelimoottorin kanssa, mutta se on myös tällä hetkellä parhaiten tuettu. [8]

Canvaksen oman 2D Contextin lisäksi selaimilla on omia Context -rajapintoja. Yleisesti ottaen nämä ovat huomattavasti laajempia ja tehokkaampia, mutta niiden tuki toisilla selaimilla on olematonta. Esimerkiksi Operalla on kehitteellinen 2D-game Context, joka on suunnattu erityisesti pelikehitykseen. WebGL on myös saanut paljon tukea eri selainten välillä. Se on Canvaksen Context-rajapinta, joka mahdollistaa grafiikkakiihdyttimen käytön piirrossa ja mahdollistaa 3D-piirtämisen Canvas-elementille. [8]

4 TYÖN TOTEUTUSSUUNNITELMA

Ohjelmistoprojekteilla on tunnetusti tapa paisua, varsinkin jos ei ole tarkkaa tietoa mitä lopullinen projekti pitää sisällään. Tämän takia huolellinen suunnittelu on yleensä avainasemassa onnistuneissa projekteissa. Myös hyvät kehitystyökalut edesauttavat projektin etenemistä. Näitä toimintatapoja on pyritty omaksumaan ja pitämään yllä projektia kehitettäessä.

4.1 Kehitysympäristö

Pääasiallisena kehitysympäristönä on työssä käytetty Googlen Chrome -selainta ja projektia on pyöritetty paikallisella palvelimella. Chromen JavaScript moottori on hyvin tehokas joten se edesauttaa projektin alkuun pääsemisessä, jättäen selainkohtaisen optimoinnin työn jälkiosaan. Koska JavaScriptiä ei käännetä, niin sen tuottamien virheiden paikantaminen on hankalaa. Tähän käytettiin hyväksi Chromen kehittäjätyökaluja ja sisäänrakennettua *console.log*-funktiota, jolla voidaan kirjoittaa viestejä kehitystyökalun konsoliin.

Ohjelmointiympäristönä käytettiin SublimeText 2 -tekstieditoria [9], josta löytyy hyvä JavaScriptin lähdekoodin tarkistaja. Tämä on tärkeää sillä kun ohjelmaa ei käännetä, huomataan virheet vasta mahdollisesti suoritusvaiheessa - pahimmassa tapauksessa vasta valmiissa projektissa puoli vuotta julkaisun jälkeen. Lähdekoodin tarkastaja helpottaa huomattavasti työtä, koska virheet voidaan huomata jo kirjoitusvaiheessa, ilman että tarvitsee suorittaa ohjelmaa.

Versioiden jatkuvuuden ja hallinnan kannalta projekti on laitettu Git-versiohallinnan alle [10]. Tämä edesauttaa pelimoottorin kasvaessa uusien ominaisuuksien lisäämistä projektiin hajottamatta nykyistä järjestelmää ja pitämällä eri versiot tallella. Sen avulla voi myös erottaa kokeelliset kehityshaarat itse projektin päähaarasta, jolloin pelimoottorin toiminnan rikkovien suurien virheiden ilmaantuminen on helpommin estettävissä.

4.2 Kehityksen eri vaiheet

Vaikka ajatus pelimoottorin toteuttamisesta HTML5 Canvas -elementillä oli työn aloittamisen aikaan uusi, oli itse pelimoottorin arkkitehtuuria hahmoteltu jo pidempään toisten projektien merkeissä. Tämä otti taakkaa pois tutkimusvaiheesta, sillä näin säästyttiin moottoreiden eri arkkitehtuurien vertailulta ja valitsemiselta.

Työn alussa tietoperusta Canvas-elementin toiminnasta sekä JavaScriptin oliopohjaisista ominaisuuksista ei ollut erityisen laaja. Tästä syystä aivan aluksi tutustuttiin siihen mitä tietoa oli saatavilla HTML5: n tarjonnasta pelialustana. Myös ympäristön takia oli tarve syventyä monimuotoisemman olio-pohjaisen JavaScriptin käyttöön, sillä varsinkin pelimoottoreissa oliot ovat erittäin käytännöllisiä. Siitä huolimatta että itse pelimoottorin arkkitehtuuri oli hyvin pitkälti selvillä, tarvitsee silti sen suunnittelutyössä ottaa huomioon ympäristön rajoitukset sekä vaatimukset.

Tutkimisen jälkeen aloitettiin pelimoottorin suunnittelu rajaamalla ja hahmottelemalla sen tarvittavat toiminnot. Jo projektin alussa oli selvää, että pelimoottori tulee olemaan hyvin suppea, jotta se saadaan myös valmiiksi työn puitteissa. Tämän takia rajaaminen on hyvin tärkeää, sillä liian kunnianhimoiset projektit jäävät helposti toteutumatta, ja toisaalta liian suppeista ei jää tarpeeksi työn aiheeksi. Suunnittelua jatkettiin hahmottamalla keskeiset osat arkkitehtuurin kannalta ja sovittamalla ne toimimaan JavaScriptin kanssa.

Kun pelimoottorin pääosien toiminta oli hahmoteltu, niin sen sijaan että koko pelimoottori olisi suunniteltu loppuun vesiputousmallin tavoin, päätettiin lähestyä projektia ketterällä menetelmällä [11]. Tarkoituksena oli saada ensin pelimoottori pyörimään ja sen jälkeen lisätä siihen ominaisuuksia pienissä, helposti ymmärrettävissä osissa. Tämä helpottaa kehittämistä varsinkin ennalta tuntemattomassa ympäristössä, jolloin itse suunnittelutyön kuormaa jaetaan kehitysvaiheeseen ja kokonaiskuva projektista paranee asteittain samalla kun ympäristö tulee tutummaksi.

5 PELIMOOTTORIN MÄÄRITTELY

5.1 Ympäristö

Yleensä pelimoottorit joutuvat hoitamaan paljon alhaisen tason asioita, kuten muistienhallintaa, ja olemaan selvillä käyttöjärjestelmän tilasta sekä sen toiminnoista. Koska tässä työssä kehitetyssä pelimoottorissa käytetään kielenä JavaScriptiä ja tulkkina selaimia, päästään tältä osin hyvin pienellä työllä. Selain toimii hiekkalaatikkona pelimoottorille ja sen ei teoriassa tarvitse ottaa kantaa käyttöjärjestelmään tai sen tiloihin missä se pyörii. Eri selaimissa tosin saattaa olla eroavaisuuksia eri JavaScriptin funktioiden toiminnallisuuksien kanssa. Tämän takia rakennettavassa pelimoottorissa pyritään pysymään mahdollisimman standardissa JavaScriptissä, eli välttämään ECMAscript ohjeistuksesta puuttuvia ominaisuuksia, jotka voivat poiketa selainten välillä.

Ympäristön helppous ei tule ilman kääntöpuolta. Koska selain on itse ohjelma, joka pyörii käyttöjärjestelmässä, syö se itsessään jo resursseja. Sen lisäksi dynaamisena komentosarjakielenä JavaScript ei voi päästä tulkitun, valmiiksi käännetyn kielen kanssa samaan tehokkuuteen. Koska pelimoottori pyörii selaimen omassa virtuaalikoneessa, on sen suorituskyky lisäksi suoraan riippuvainen selaimen JavaScript-moottorin suorituskyvystä. JavaScript moottoreiden tehokkuus varsinkin eri tehtävien välillä voi vaihdella suuresti selaimissa ja laitekokoonpanoissa. Sen takia pelimoottorin sulavan toiminnan takaaminen jokaisessa selaimessa ja vähätehoisemmissa koneissa on haastavaa sekä vaatii testausta ja optimointia.

5.2 Vaatimukset

Pelin sisältö ja ulkoasu eivät ole olennainen osa toteutusta; tarkoituksena on vain saada aikaan toimiva pelimoottorin runko jonka ominaisuuksia voidaan laa-

jentaa jatkossa. Pelimoottorin ja siinä pyörivän pelin prototyypin avulla voidaan testata HTML5 tekniikan potentiaalia pelikehityksessä.

Pelimoottorin keskeisiin ominaisuuksiin kuuluu oma prosessienhallinta, tapahtumienhallinta, karkea resurssienhallinta, kyky piirtää Canvakselle sekä ladata ja toistaa äänitiedostoja. Se mahdollistaa myös hyvin karkean käyttöliittymän tekemisen, joka rajoittuu näppäinkomentoihin.

Pelimoottorin toiminnoista on rajattu pois paljon sellaista mitä yleensä odotettaisiin löytyvän. Näihin ominaisuuksiin kuuluvat fysiikkamoottorin ja tekoälymoottorin sisällyttäminen osaksi pelimoottoria. Prototyypissä ei ole käytännössä lainkaan tekoälyä ja fysiikkamoottorin sijalla on kevyt olioiden yhteentörmäyksen tarkistusfunktio sekä näennäisen painovoiman vaikutuksen lisääminen pelaajalle. Pelimoottori ei myöskään tue useampaa kenttää tai pelin tilanteen pysyvää tallentamista missään muodossa. Myös monisäikeisyys on jätetty pois, sillä pelimoottorilla ei tässä muodossa ole mitään niin laskentaintensiivistä tehtävää, jossa tarvittaisiin käyttää hyväksi säikeitä.

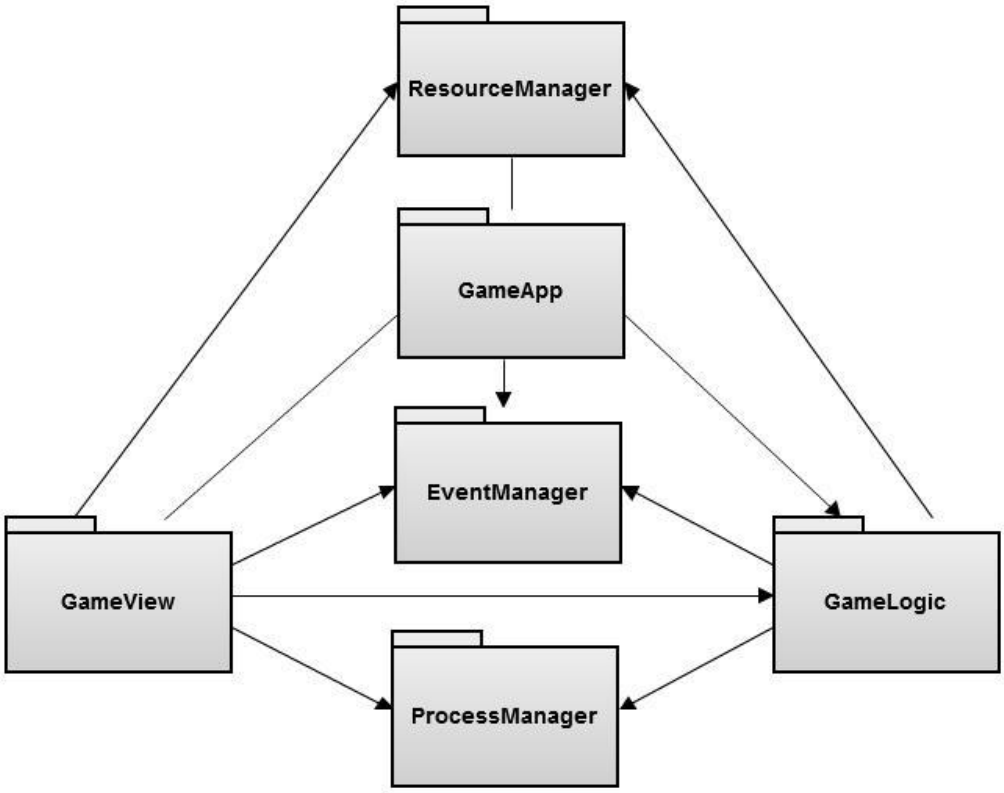
Itse prototyyppipeli on sivusta päin kuvattu loputtomasti jatkuva tasohyppely, jossa pelattava kenttä luodaan yksinkertaisella satunnaisgeneraattorilla. Siinä on käytännössä kolme tilaa, joissa se voi olla. Se voi olla käynnissä, pysäytetty hetkellisesti tai alkuvalikossa odottamassa käyttäjän syötettä. Pelin ideana on pysyä kentällä, ja jos pelaaja putoaa ruudun ulkopuolelle, palataan takaisin alkuvalikkoon.

6 PELIMOOTTORIN RAKENNE

Pelimoottorin rakenteiden taustalla on ajatuksena tehdä siitä helposti laajennettava, jotta monipuolisempia ominaisuuksia on helpompaa lisätä jälkikäteen. Tämän takia sen eri osat on yritetty eristää toisistaan mahdollisimman tehokkaasti.

6.1 Kerrokset

Pelimoottorin toiminnot on käytännössä jaettu kolmeen kerrokseen: *GameApp*, *GameLogic* ja GameView. Jokaisella osalla on tietty rooli, joilla pyritään erottamaan tietyntyyppiset osa-alueet pelimoottorista helpommin hallittaviin ja eristettyihin lohkoihin. Kuviossa 1 havainnollistetaan näiden kerrosten väliset riippuvuudet toisistaan, sekä miten ne käyttävät managereita (ks. luku 6.2) keskustelemaan keskenään.



Kuvio 1 Ohjelman pääosien väliset riippuvuudet toisistaan sekä managereista

Arkkitehtuurissa on myös pyritty siihen, että eri osat olisivat riippumattomia toisistaan ja teoriassa sekä ideaalitilanteessa yhteen tehdyt muutokset eivät vaatisi muiden osa-alueiden muokkausta. Käytännössä tämä on kuitenkin varsinkin suurten muutosten kohdalla erittäin vaikeaa, sillä osa-alueet eivät yksinään muodosta eheää kokonaisuutta ja luottavat toisiinsa sekä niistä löytyviin toiminnallisuuksiin. Myös kuvio 1:ssä olevat managerit ovat kriittisessä roolissa kerrosten toiminnan kannalta.

6.1.1 GameApp

Kolmesta osasta *GameApp* on ehkäpä tärkein pelimoottorin toiminnan kannalta, vaikka se onkin pienin. Sen tarkoituksena on toimia ympäristön ja pelimoottorin välisenä abstraktiona. Jos pelimoottori joskus käännettäisiin toiselle alustalle

kuin Canvas -elementille, niin ideaalitilanteessa koko pelimoottorista tarvitsee tehdä vain muutoksia tähän osaan. Kuviossa 2 on kuvaus *GameApp* oliosta ja sen ominaisuuksista. Sen attribuutteihin kuuluu viittaus HTML5:n Canvas elementtiin sekä *AudioManager* olioon joka hoitaa äänielementtien lisäämisen ja poistamisen DOM-puusta. GameApp on ainoa osa pelimoottoria joka keskustelee suoraan ympäristön kanssa.

Kuvio 2 *GameApp* kerroksen rakenne

Koska kuitenkin kyse on komentosarjakielellä tehdystä sovelluksesta, tarvitsee se ympäristöltä toimiakseen silti virtuaalikoneen joka tulkitsee JavaScriptiä samalla tavalla kuin selaimet. Vaikka ympäristön piirtokutsut, äänien toistamiset ja käyttäjän syötteiden lukemiset hoidetaan tässä kerroksessa, luottaa koko pelimoottori esimerkiksi JavaScript-moottorin olemassaoloon.

Abstraktointi

Yksi *GameApp*in kulmakivistä on hoitaa ympäristön ja pelimoottorin erotus toisistaan. Tämä tarkoittaa että vain *GameApp* keskustelee järjestelmän kanssa ja välittää käskyt eteenpäin muille pelimoottorin osille. Tämä toimii myös molempiin suuntiin. Esimerkiksi kun pelimoottorin osa tarvitsee piirtää jotain Canvakselle, se ei kutsu itse suoraan piirtofunktiota vaan välittää käskyn *GameApp*in piirtofunktioon, joka hoitaa Canvaksen Contextin piirtofunktion kutsumisen. Tämä tehdään siitä syystä, että jos jatkossa tulee tarve vaihtaa esimerkiksi Canvas-elementin Contextia, niin periaatteessa tarvitsee muutos tehdä vain *GameApp*in piirtofunktioon, eikä jokaiseen olioon, joka sitä kutsuu. Käytännössä tosin pitää ottaa huomioon, että eri Contextit tai mahdollisesti muut ympäristöt saattavat tarvita toiset parametrit kuin mitä nyt oliot antavat, joten asia ei ole niin yksioikoinen.

Canvas-elementin lisäksi abstraktiota on toteutettu ottamalla käyttäjän syötteet vastaan ja lähettämään ne eteenpäin pelimoottorin sisäisinä komentoina. Tällöin jos ympäristö muuttuu pois selaimista, tarvitsee vain muuttaa *GameApp*in syötteen vastaanottavat toiminnot koskematta pelimoottorin muihin osiin.

Tehtävät

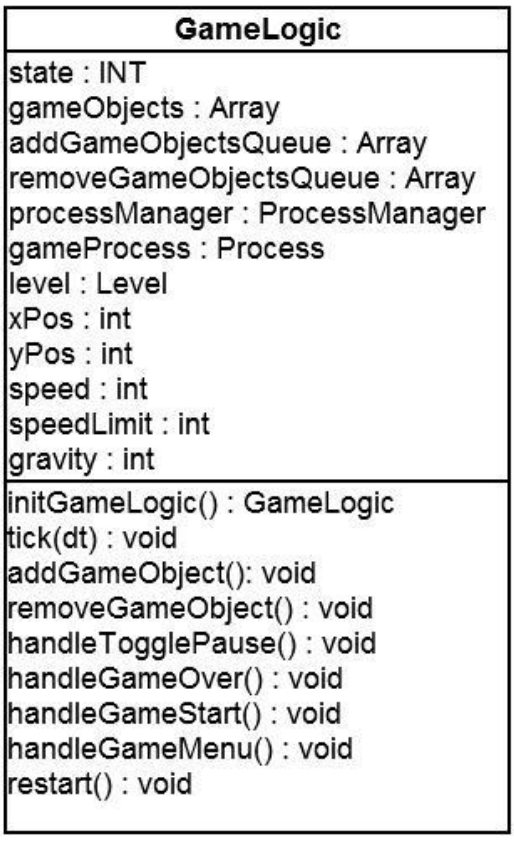
Sen lisäksi että *GameApp* toimii ympäristön ja pelimoottorin välisenä rajapintana, toimii se myös pelimoottorin käynnistämisen alustajana. *GameApp*in käynnistämisfunktiossa *initGameApp,* huolehditaan pelimoottorin muiden pääosien luonnista ja niiden asettamisesta globaaleihin muuttujiin. Vaikka yleensä globaaleja muuttujia paheksutaan, se on tässä tapauksessa hyvä kompromissi nopean saatavuuden ja potentiaalisen nimiavaruuden saastuttamisen kannalta. Moniin pelin osiin on hyvä päästä suoraan käsiksi, ilman että tarvitsisi mennä *GameApp*in kautta.

Koska *GameApp* huolehtii ympäristön kanssa keskustelusta, täytyy sen huolehtia elementtien luonnista, poistosta ja etsimisestä DOM-puusta. Vaikka tämän

työn osalta pelimoottori ei itse luo Canvas-elementtiä dynaamisesti, niin tarvitsee sen silti vielä etsiä se dokumentista ja asettaa siitä viittaus omaan muuttujaansa, jotta sitä on helpompi käyttää. Myös äänijärjestelmä toimii uusien HTML5-elementtien kautta. Vaikka *GameApp* ei itse määritä, mitä ääniä ladataan, se huolehtii silti *audio*-elementtien lisäämisestä ja poistamisesta DOM:sta. Myös pääsilmukka pyörii tässä osassa pelimoottoria.

6.1.2 GameLogic

GameLogic-kerros vastaa pelimoottorin ja siinä pyörivän pelin säännöistä ja tilasta. Sen päätehtävänä on alustaa pelimekaniikan prosessienhallinta ja pitää kirjaa kaikista pelissä olevista olioista. Kuviossa 3 on esitetty sen rakenne.

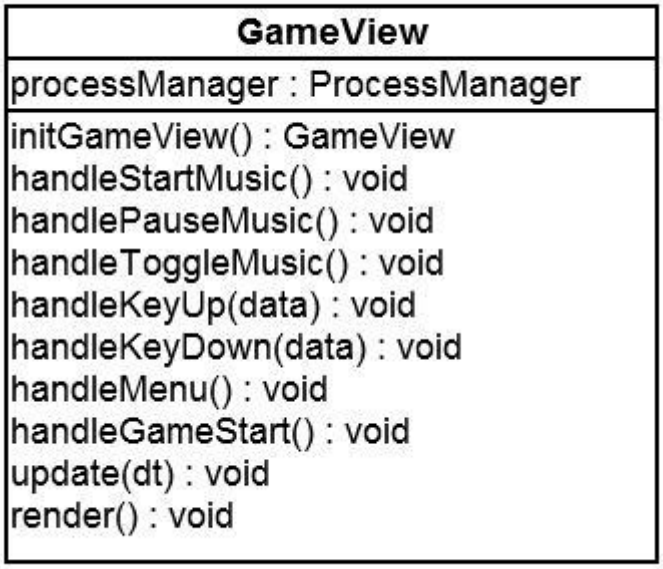


Kuvio 3 *GameLogic* kerroksen rakenne

GameLogic sisältää metodeja uusien olioiden lisäämiseksi sekä vanhojen poistamiseksi pelimaailmasta. Se myös pitää ylläpitää pelin tilaa. Kerroksen *tick*-metodi on suuressa roolissa. Sitä kutsutaan *GameApp*-kerroksen pääsilmukassa. Tässä metodissa tarkastellaan missä tilassa pelimoottori on tällä hetkellä ja, jos peli on käynnissä, päivitetään pelilogiikan prosessienhallintaa. *GameLogic* ei itse piirrä mitään Canvas-elementille tai soita ääniä. Se ei myöskään ota vastaan suoraan käyttäjän käskyjä. Se vain yksinkertaisesti pitää yllä pelin sääntöjä, tilannetta ja olioiden olemassaoloa pelimaailmassa. Tähän kerrokseen myös kuuluisi fysiikka- ja tekoälykomponentit, mikäli sellaiset olisivat kuuluneet tämän työn rajaukseen.

6.1.3 GameView

GameView-kerroksen tarkoituksena on toimia käyttäjän ikkunana pelimoottoriin. Se päättää mitä näytetään käyttäjälle, tai mitä käyttäjä kuulee. Se myös kuuntelee käyttäjän antamia näppäinkomentoja *EventManager*in kautta (ks. luku 6.2.3), päättää mitä niillä tehdään ja välittää ne eteenpäin logiikkakerrokselle tapahtumakutsuina. Myös käyttöliittymä elää tässä kerroksessa. Koska pelimoottorin pitää pystyä piirtämään Canvakselle jokaisessa vaiheessa ja sen täytyy myös pystyä reagoimaan käyttäjän syöttöihin, vaikka peli olisikin pysäytetty, täytyy *GameView* -kerroksen olla riippumaton *GameLogic* -kerroksen tilasta. Tästä johtuen sillä on myös oma prosessienhallinta–olionsa. *GameView*:n päivittämien on myös erotettu pääsilmukassa omaan kutsuunsa. Hieman ehkä harhaanjohtavasti pääsilmukassa kutsutaan *GameApp*:n *render*-funktiota. Vaikka *render*-funktion päätarkoituksena on hoitaa piirtäminen Canvas-elementille, kutsuu se myös *GameView*in *update*-funktiota niiden oman *render*-funktion lisäksi. Kuvio 4 esittelee sen rakennetta.



Kuvio 4 *GameView* kerroksen rakenne

Suurin rakenteellinen ero *GameView*in ja muiden kerrosten kesken on, että niitä voi teoriassa olla useita. Sen sijaan että *GameView*-kerrokseen viitattaisiin globaalin muuttujan avulla, tallennetaan viittaukset eri *GameView*-olioista *GameApp*issa sijaitsevaan taulukkoon. Käytännössä tosin kyseinen toiminnallisuus ei kuulu tämän työn sisältöön.

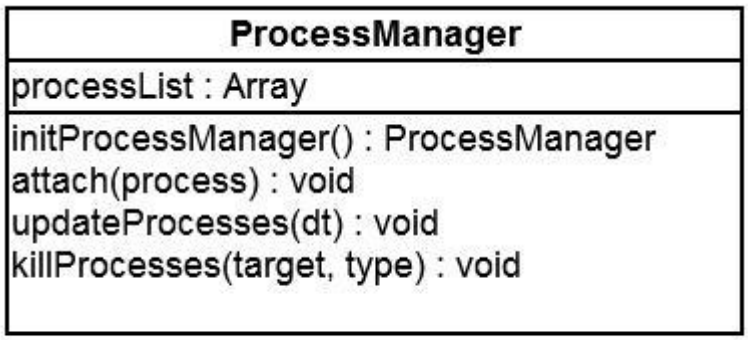
6.2 Managerit

Pelimoottorin eri kerrosten pitäminen erillään vaatii, että ne pystyvät kommunikoimaan keskenään. Tähän tarkoitukseen siitä löytyy kolme eri manageria. Vaikka managereita on sama määrä kuin kerroksia, eivät niiden tehtävät liity suoraan kerrosten tehtävään, vaan jokaisella managerilla on kriittinen rooli pelimoottorin muun toiminnan kannalta. Niiden päätehtävänä on hoitaa yksittäisten tapahtumien tai asioiden päivittämistä ja huolehtia resurssien lataamisesta sekä niiden tilasta. Niiden tarkoituksena on jättää itse pelimoottorin pääosat huoleh-

timaan yleisimmistä korkean tason asioista ja auttaa niitä keskustelemaan toistensa kanssa.

6.2.1 ProcessManager

Kuten *ProcessManager*in nimestä voi päätellä, *ProcessManager* huolehtii prosessien hallinnasta ja päivittämisestä. Jokaisesta pelin tapahtumaketjusta, joka kestää enemmän kuin yhden päivityssyklin, voidaan luoda prosessi. Esimerkiksi jos pelaajaa kuvaava olio hyppää tai törmää esteeseen, voidaan siitä seuraavaa tapahtumaketju toteuttaa prosessina, joka liitetään prosessimanageriin. Prosessit voivat olla jatkuvia, jolloin niiden lopettamisesta huolehtivat niiden tekijät tai ne voivat olla tietyn kestoisia, jolloin ne itse huolehtivat siitä, että kun niiden tehtävä on suoritettu, ne poistavat itsensä *ProcessManager*ista. Prosesseja ei myöskään suoraan poisteta ProcessManagerista. Kun halutaan lopettaa prosessi, annetaan sen *isDead* arvoksi *true*. Tällöin jos ProcessManagerin *updateProcesses*-funktiossa on prosessi merkitty kuolleeksi, poistetaan se managerista päivityksen sijaan.



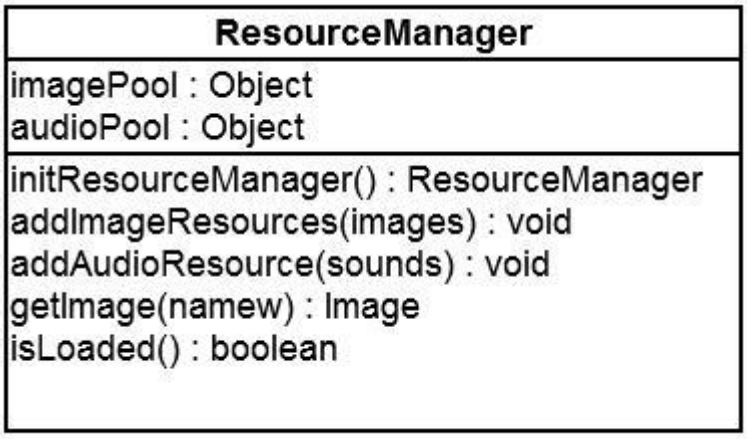
Kuvio 5 Prosessimanagerin rakenne

Kuviosta 5 on esitetty prosessimanagerin metodit ja sen ainoa attribuutti. *ProcessList* sisältää kaikki prosessimanageriin *attach*-metodin kautta annetut prosessit. Nimensä mukaisesti *killProcesses* merkitsee prosessin kuolleeksi. Se ottaa parametriksi prosessin kohteen ja tyypin, jos halutaan tappaa tietty pro-

sessi. Jos mitään parametreja ei anneta, tapetaan kaikki oletusarvon omaavat prosessit.

6.2.2 ResourceManager

Peliin kuuluu paljon resursseja, joita siihen ei voi tai kannata toteuttaa ohjelmoimalla. Näihin kuuluu kuvat ja äänet. Resurssimanagerin tehtävänä on pitää yllä listaa pelin tarvitsemista kuva ja äänitiedostoista sekä niiden tilasta.



Kuvio 6 Resurssimanagerin rakenne

Kuviossa 6 esitellään resurssimanagerin rakenne. Kuvatiedostot säilytetään siinä näytetyssä *imagePool*-oliossa. Ne annetaan resurssimanagerille *addImageResources*-funktiolle taulukkona, jonka jokainen alkio on olio, jolla on kaksi muuttujaa. Toinen niistä on kuvan nimi ja toinen sen sijainti. Kuvat annetaan oliona sen takia, että kuvaresursseihin pääsee käsiksi ja vältytään samojen kuvien lataukselta, jos samannimiset kuvat sattuvat jo olemaan resurssimanagerissa.

Äänitiedostot annetaan resurssienhallinnalle parametreina samassa muodossa käyttäen *addAudioResrources*-funktiota, mutta audioresurssin oliolla on yksi lisämuuttuja, *poolSize*, jonka merkitykseen palataan myöhemmin. Audioresurssit eroavat kuvista sen verran, että ne tarvitsevat toimiakseen oman HTML5 *audio* -elementtinsä DOM-puussa. Uuden elementin, tai toisin sanoen ääni-

raidan, lisäys tapahtuu *GameApp*in rajapinnan kautta, jotta resurssimanageri saadaan eristettyä ympäristöstä ja se ei itse muokkaa suoraan DOM-puuta. Tosin tämän eristyksen toiminta ei ole täydellinen, koska audioPool-muuttuja tallentaa viittauksen itse *audio*-elementtiin, jotta sen toimintoihin olisi helpompi ja käytännöllisempi saatavuus.

Usean äänen yhtäaikainen toisto

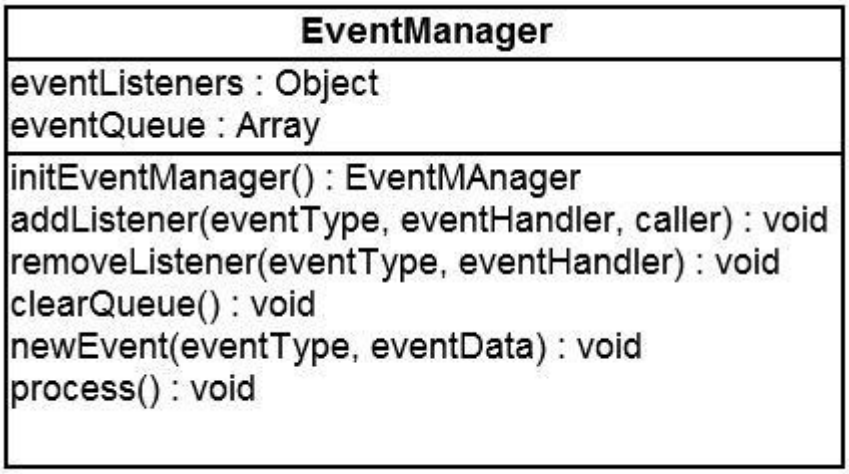
HTML5:n *audio*-elementissä on yksi ongelma. Se ei pysty soittamaan ja pitämään ladattuna kuin yhtä ääntä kerrallaan. Tämä ei olisi ongelma, jos haluttaisiin esimerkiksi vain soittaa musiikkia, mutta koska peleissä yleensä tarvitaan monia päällekkäisiä ääniä, tarvitaan ääniraitoja enemmän kuin yksi. Yksinkertainen lähestymistapa olisi tehdä jokaiselle äänitiedostolle oma audio-elementtinsä. Tämä voi alkuun vaikuttaa riittävältä ratkaisulta, mutta ongelmaksi muodostuu, jos jonkin äänen tarvitsee toistua uudestaan ennen kuin vanha on lopettanut. Pelimoottorissa ongelma on ratkaistu määrittelemällä jokaiselle äänelle kuinka monta ääniraitaa, tai käytännössä audio–elementtiä, sillä on. Tämä määritellään aikaisemmin mainitulla *poolSize*-parametrilla. Ääniraitoihin esiladataan valmiiksi äänet ja kun halutaan toistaa ääntä, resurssienhallinta etsii a*udioPool* -muuttujasta ääniraidan ja tarkistaa onko se käytössä. Se käy ääniraitataulukkoa niin kauan läpi kunnes se löytää raidan joka ei ole käytössä tai taulukko loppuu. Jos yhtään ääniraitaa ei ole käytössä, luo se uuden ääniraidan. Tässä täytyy olla myös varovainen, sillä liian moni *audio*-elementti saattaa hidastaa peliä ja syödä tarpeettomasti resursseja. [8]

6.2.3 EventManager

EventManager on pelimoottorin viestikanava. Se mahdollistaa eri osien välisen kommunikaation ilman, että niiden tarvitsee tietää toisistaan. Jos mietitään skenaarioita missä *GameApp* kuuntelee järjestelmän antamia käyttäjän syöttöjä ja sen tarvitsee välittää tieto eteenpäin, koska sen tehtävänä ei ole päättää mitä niiden käskyjen pohjalta tehdään. Perinteinen ratkaisu olisi kutsua rajapinnan

kautta *GameView*-kerroksen *keyDown*-funktiota, joka voi sitten päättää mikä näppäin on pohjassa ja mikä on siihen näppäimeen sidottu toiminto. Tämä on huono ratkaisu siinä mielessä, että *GameApp* joutuu luottamaan, että *GameView*illä on olemassa *keyDown*-funktio. Jotta tätä eri osien välistä riippuvuutta ei pääse syntymään, eristetään käskyt omiksi tapahtumiksi.

Vastaavasti kyseinen toiminto toteutetaan *EventManager*in avulla seuraavasti. Kun järjestelmä alustetaan, niin *GameView* lisää *EventManager*iin tapahtumakuuntelijan, *eventListener*in. Tämä tapahtuu *EventManager*in *addListene*-funktion kautta (Kuvio 7). Se ottaa kolme parametria, tapahtuman tyypin, tapahtuman käsittelijäfunktion ja olion, jolla kutsutaan käsittelijäfunktiota. Kun lisätään uutta tapahtumakuuntelijaa, ei viimeinen parametri ole pakollinen. Sitä tarvitaan vain jos käsittelijäfunktion tarvitsee päästä käsiksi oman olionsa tietoihin. Tämän raportin JavaScript osassa mainittiin, että funktion *this*-attribuutti viittaa eri asioihin riippuen miten funktiota kutsutaan ja koska nyt käsittelijäfunktiota kutsutaan toisen olion kautta, ei sen *this*-attribuutti viittaa käsittelijäfunktion omaan olioon, ellei käsittelijäfunktiota kutsuta sen *call*-metodin kautta antaen oikean olion parametrina.



Kuvio 7 Tapahtumamanagerin rakenne

Pelkkä tapahtumakuuntelija vain vastaanottaa tapahtumia. *EventManager*:n kautta voidaan luoda uusia tapahtumia jonoon. Tapahtuma olio koostuu kahdesta attribuutista. Tapahtuman *type* -attribuutti määrittää tyypin jolla tapahtuma

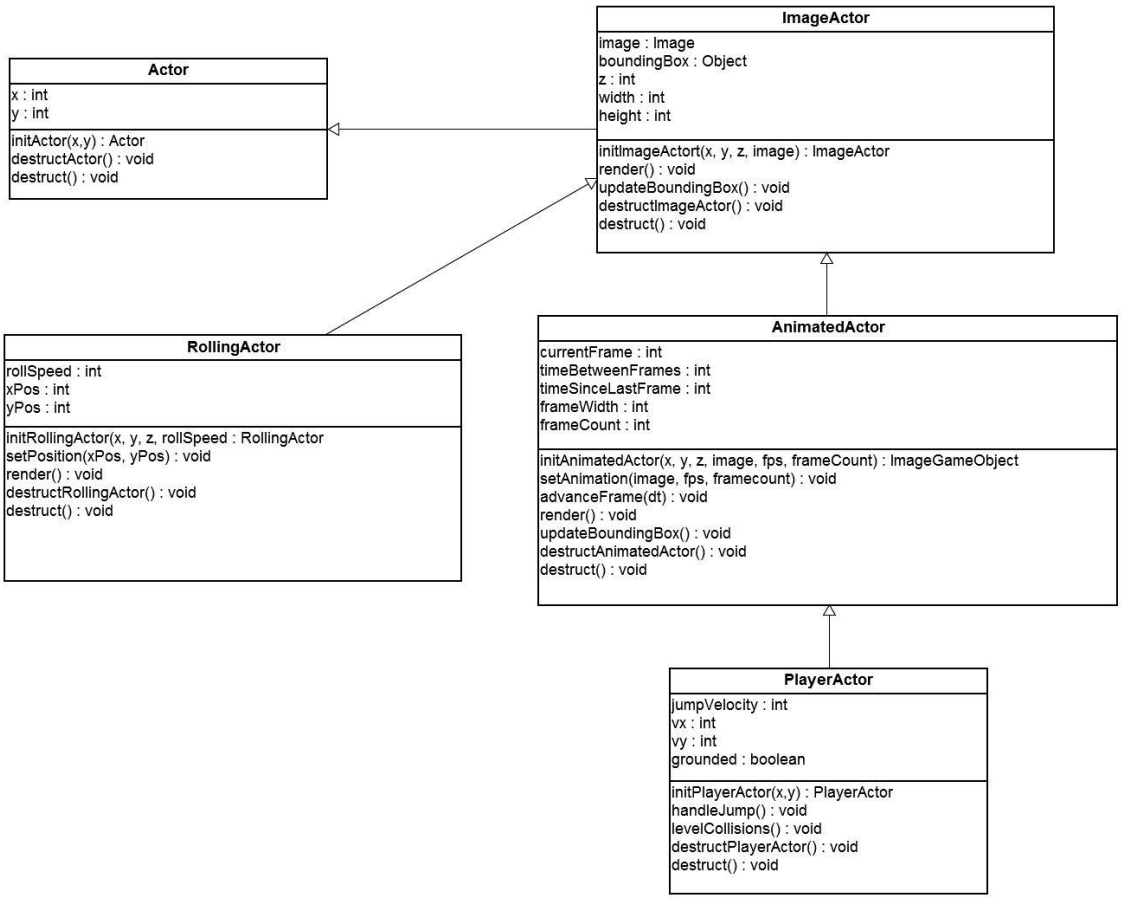
yhdistetään tapahtumakuuntelijoihin. Vapaaehtoinen *data* -attribuutti sisältää mahdolliset parametrit, joita voidaan liittää tapahtumaan. Aikaisemman esimerkin mukaisesti luotaisiin *GameView*-kerrokseen *handleKeyDown*-tapahtumakäsittelijä ja liitettäisiin se *KEY_DOWN*-tapahtumatyypin kanssa *EventManager:*n uudeksi tapahtumakuuntelijaksi. Tämän jälkeen *GameApp*-kerros voi ottaa ympäristöltä vastaan näppäinkomentoja ja luoda niistä uuden tapahtuman *EventManager*in *newEvent*-metodilla. Hyvänä puolena on se, että *GameApp*in ei tarvitse tietää mikä pelimoottorin osa tarvitsee tai käyttää sitä tapahtumaa ja taas *GameView*-kerroksen ei tarvitse tietää mistä tapahtuma tulee.

6.3 Pelimoottorin muut osat

Vaikka eri kerroksilla ja managereilla pääsee pitkälle, eivät ne vielä yksinään riitä rakentamaan peliä. Tarvitaan myös oliot jotka muodostavat pelissä elävät asiat. Näitä ovat itse pelattava hahmo, pelattava taso ja siinä olevat oliot, kuten alusta jolla pelattava hahmo yrittää pystyä sekä taustakuvat. Myös eri prosessit muodostuvat omista olioistaan.

6.3.1 Pelioliot

Pelimaailmassa elävät asiat ovat funktio -olioita, joiden prototyyppipohjana toimii *Actor*-olion prototyyppi. Kuviossa 8 esitellään eri pelimaailman olioiden rakenne ja niiden perintäsuhteet.



Kuvio 8 Pelimaailman oliot; Kantaoliona on *Actor*, josta johdetaan *ImageActor*. *ImageActori*:sta peritään *RollingActor* ja *AnimatedActor*, jos taas peritään itse pelin oma olio, *PlayerActor*

Actor-olio on hyvin yksinkertainen ja sen toiminnallisuuksiin kuuluu vain omat *x*- ja *y* -koordinaattien, sekä syvyyttä piirtoalustalla kuvaavan *z*-attribuutin tallentaminen. Metodeina oliolla on *initActor* -funktio, joka toimii olion tietojen alustajana ja *destructActor*-funktio joka toimii sen tuhoajana (ohjelma 4). Kuviosta 8 huomaa että nimeämiskäytäntönä sekä alustaja ja tuhoajafunktiot lainaavat loppuosan nimestään itse olion nimestä. Tämä ei ole JavaScriptin ominaisuus, vaan lähinnä sääntö tämän pelimoottorin arkkitehtuurissa. Nimeämiskäytäntönä kaikilla tästä johdetuissa luokissa noudatetaan samaa periaatetta. Sen lisäksi, että tiedetään missä funktiossa ajetaan olion attribuuttien alustus, on nimeämiskäytännöllä myös toinen syy. JavaScript ei itsessään tue perittyjen olioiden vanhempien omien metodien kutsua. Tämä voidaan kiertää käyttämällä jokaisessa oliossa omaa alustus ja tuhoamisfunktioita joiden avulla voidaan helposti

kutsua myös sen funktion alustusta, josta olio on johdettu (ohjelma 5). Tämä myös mahdollistaa tuhoamisfunktion ketjuttamisen siten, että olion prototyypille määritellyssä tuhoamisfunktiossa hoidetaan vaan olion omien attribuuttien tuhoaminen ja sen perityissä tuhoamisfunktioissa hoidetaan perittyjen attribuuttien tuhoaminen. Vertailemalla ohjelmia 4 ja 5 saa hyvän käsityksen miten perintä on hoidettu käytännössä muidenkin luokkien osalta.

Tutkimalla lähdekoodiotteita käy ilmi että olioilla on kaksi tuhoamisfunktiota. Tarkoituksena taustalla on antaa kaikille *Actor*-oliolle yhteinen tuhoamisfunktio, jota voidaan kutsua ilman olion oman tuhoamisfunktion nimen tietämistä. Esimerkiksi jos suuri määrä eri olioita on tallennettu taulukkoon ja haluamme poistaa ne kaikki. Koska niillä on samanniminen tuhoamisfunktio, voimme yksinkertaisesti vain iteroida taulukon läpi ja kutsua jokaisen olion *destruct*-metodia, joka taas olion sisällä kutsuu sen omaa tuhoamisfunktiota.

Ohjelma 4 *Actor* olion lähdekoodi. *Actor* toimii pohjana muille *Actor* luokille

```
function Actor()
{
    /*  The position on the X axis
    @type Number */
    this.x = 0;

    /*  The position on the X axis
    @type Number */
    this.y = 0;

    /* If this object is alive, so we don't remove it */
    this.alive = null;
```

Ohjelma 4 (jatkuu)

```
    /**
     Initialises this object
     @param x @type Number
     @param y @type Number */
     this.initActor = function(x, y) {
     this.x = x;
     this.y = y;
     this.alive = true;
     return this;
}
     /** Destroy the actor, set all of it's properties to null.  */
     this.destructActor = function(){
         this.x = null;
         this.y = null;
         this.alive = null;
     }
     /** This will be called if the we want to destroy the actor */
     this.destruct = function() {
         this.destructActor();
     }
}
```

Actor-olio ei yksinään ole kovin käytännöllinen, mutta se toimii pohjana muille monipuolisimmille olioille. *Actor*-oliosta johdettu *ImageActor* lisää oliolle attribuutteina viitteen kuvaan sekä *boundingBox*-olion, jossa määritellään nelikulmio, joka kuvaa olion kokoa ja rajoja pelimaalimassa. *ImageActor* sisältää *render*-metodin, jossa kutsutaan *GameApp*in piirto funktioita jolle annetaan viite piirrettävästä kuvasta sekä kuvan sijainti. *ImageActor*-oliosta johdetaan edelleen kaksi erikoistunutta oliota; *AnimatedActor* sekä *RollingActor*.

Ohjelma 5 *ImageActor* olion lähdekoodi, josta käy hyvin ilmi miten perintä hoidetaan käytännössä *Actor*-oliosta.

```
function ImageActor() {
    /** The "depth" of the image
    @type Number */
    this.z = 0;

    /** The image that is assigned to this object
    @type Image */
    this.image = null;

    /** The invisible box we are using to handle the collisions
    @type Object */
    this.boundingBox = {};

    /** The width that the final image will take up
    @type Number */
    this.width = 0;

    /** The height that the final image will take up
    @type Number*/
    this.height = 0;

    /**
        Initialises this object
       @param x @type Number
        @param y @type Number
       @param z @type Number
        @param image @type Image
    */
```

Ohjelma 5 (jatkuu)

```
    this.initImageActor = function(x, y, z, image)     {
        this.initActor(x, y);
        this.z = z;
        this.image = image;
        this.width = image.width;
        this.height = image.height;
        this.updateBoundingBox();
        return this;
}

    /** Calls the gameApps drawImage method */
    this.render = function() {
        g_GameApp.drawImage(this.image, this.x, this.y);
    }

    /** Clean this object up */
    this.destructImageActor = function() {
        this.z = null;
        this.image = null;
        this.boundingBox = null;
        this.width = null;
        this.height = null;
        this.destructActor();
    }

    /*  This will be ovewritten in every inherited object, so that
        we have a generic function to call their own destructors. */
    this.destruct = function() {
        this.destructImageActor();
    }
```

Ohjelma 5 (jatkuu)

```
    /* A description of the actors size and position on the world */
    this.updateBoundingBox = function()  {
        this.boundingBox.left = this.x;
        this.boundingBox.top = this.y;
        this.boundingBox.right = this.x + this.width;
        this.boundingBox.bottom = this.y + this.height;
    }
}

/* Inherit the prototype of the Actor object */
ImageActor.prototype = new Actor;
```

RollingActor hoitaa toistuvien kuvien vierittämisen pelimaailmassa. Toistuvilla kuvilla tarkoitetaan kuvia, joiden alkupää ja loppupää sopivat saumattomasti yhteen, ja kuvan loppupäätä voidaan jatkaa sen alkupäällä muodostaen loputtoman silmukan. Tämän työn yhteydessä sen ainoa tehtävä on pyörittää taustakuvia, mutta käytännössä sillä on paljon muitakin käyttökohteita. *RollingActor* lisää itselleen kolme uutta attribuuttia, joista *xPos* ja *yPos* määrittelevät kuvan tämänhetkisen vierityskohdan. Kolmas *rollSpeed*-muuttuja määrittää vieritysnopeuden. Antamalla eri vieritysnopeuksia kuville, voidaan esimerkiksi taustakuvat saada kulkemaan samalla päivitystiedolla eri nopeuksia, jolla luodaan syvyysvaikutelmaa. Kolmen attribuutin lisäksi määritellään vain yksi uusi metodi; *setPosition*. Sen tehtävänä on asettaa olion sisäinen vierityskohta sekä *x*- että *y* -akselille. Tähän se käyttää *rollSpeed* -kerrointa ja annettuja vieritysparametreja, joista se laskee sisäisen vierityskohdan käyttämällä kuvan leveyttä moduluksena. Iso osa tämä olion logiikasta kuuluu sen ylikirjoittamalle *render*-metodille. Siinä tarkastellaan missä kohtaa kuvaa ollaan. Koska tässä työssä on käytetty kyseistä olioita vain taustakuviin, piirretään silmukassa kuvaa niin kauan kunnes koko Canvas on täytetty aloittaen parametrina annetusta vierityskohdasta. Jatkossa pitäisi myös pystyä määrittelemään toistuvia kuvioita pienemmille aluille

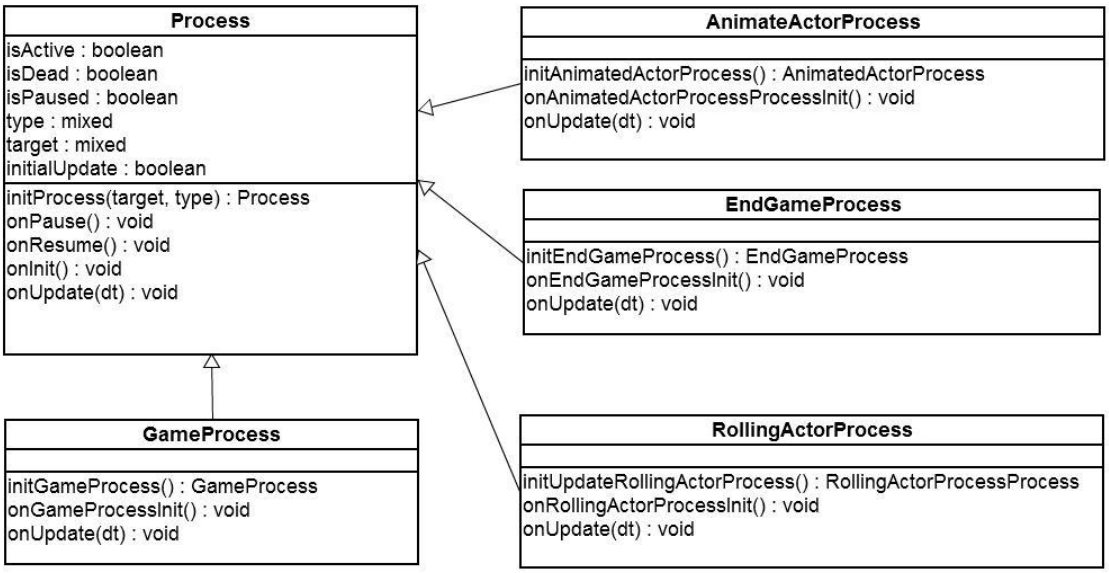
itse Canvas-elementin sisällä, mutta se toiminnallisuus on rajattu pois tästä työstä.

AnimatedActor lisää mahdollisuuden sprite-grafiikan pyörittämiseen. Tällä tarkoitetaan sitä, että olioon liitetty kuva on itse asiassa kuvasarja, joka on jaoteltu yhtä suuriin lohkoihin. Yksi lohko vastaa aina yhtä kuvaa animaatiossa. Parametreina *initAnimatedActor*-funktiolle annetaan *ImageActor*in parametrien lisäksi lohkojen määrä sekä animaation kuvataajuus, joka määrittää kuinka nopeasti animaation kuva vaihtuu sekunnin sisällä. Siihen lisättävät metodit ovat *advanceFrame*, jota käytetään päättelemään koska vaihdetaan piirrettävää kuvaa, sekä *setAnimation*, jonka tarkoituksena on määrittää animaation tiedot ja aloituskohta. Sitä voidaan myös käyttää vaihtamaan kuva sekä animaatio oliolle. Olion täytyy myös ylikirjoittaa *ImageActor*in *render*-metodi, koska piirrettäväksi annetaan vain tietty osa kuvasta.

PlayerActor-olio on johdettu *AnimatedActor*-oliosta. Se ei sinällään liity pelimoottoriin vaan siinä pyöritettävään prototyyppipeliin. Sen tarkoituksena on kuvata pelaajaa pelimaailmassa. Se lisää joukon attribuutteja jotka kuvastavat sen toimintoja, kuten kuinka korkealle ja kuinka kauan se voi hypätä. Se myös määrittää itselleen suuntavektorin, jota voidaan käyttää liikuttamaan sitä pelimaailmassa. Sillä on myös kaksi omaa metodia, joista toinen on liitetty *EventManager*in tapahtumakuuntelijaan mikä kuvaa pelaajan ”hyppy”–tapahtumaa. Toinen metodeista tarkistaa onko pelaaja turvallisesti tason päällä vai putoamassa rotkoon. Jos kyse on jälkimmäisestä, laukaisee metodi uuden tapahtuman, joka päättää pelin.

6.3.2 Prosessit

Yksittäisten prossien tarkoitus on hoitaa jatkuvaa tehtävää pelimaailmassa. Prosessimanageri huolehtii prosessien päivittämisestä ja poistamisesta, mutta itse prosessit vastaavat pelimaailman olioiden tilojen muokkaamisesta. Kuten pelimaailman olioilla, on myös prosesseilla yksi kantaolio, josta laajennetaan tarkempiin, tiettyä tehtävää hoitaviin prosesseihin kuten tulee esille kuviossa 9.



Kuvio 9 Prosessi-olioiden rakenne ja perintäsuhteet. Kaikki tällä hetkellä pelimoottorissa olevat oliot johdetaan suoraan *Process*-oliosta.

Process-olio toimii pohjana muille prosessiolioille ja sitä ei käytetä itsessään pelimoottorissa hyväksi. Se määrittää prosesseille kuusi attribuuttia. Näistä ensimmäiset kolme, *isActive*, *isPaused* ja *isDead,* kuvaavat prosessin tilaa. Jos *isActive* ei ole tosi, ei prosessia päivitetä ja jos *isDead* on tosi, poistaa prosessimanageri tämän prosessin seuraavalla päivityskierroksella. *isPaused* toimii vain tarkennuksena siitä, että prosessi on halutusti pysäytetty, mutta sitä ei haluta poistaa. Kantaprosessin *type*-attribuutti pitää sisällään prosessin tyypin. Prosessin tyyppiä voidaan käyttää hyväksi, jos halutaan poistaa tai pysäyttää kaikki tietyntyyppiset prosessit. Prosessin *target*-attribuutti pitää sisällään prosessin kohteen. Viimeinen kantaprosessin attribuutti *initialUpdate* pitää sisällään onko tämä prosessi jo käynnistetty. Prosessin ensimmäisessä päivityksessä katsotaan onko *initialUpdate*-attribuutti tosi. Jos se ei ole, niin kutsutaan prosessin *onInit*-metodia. Tässä metodissa katsotaan onko prossille annettu jokin tietty kohde jonka jälkeen lopetetaan kaikki saman kohteen ja tyypin omaavat prosessit. Tämä toimenpide tehdään sen takia, ettei päällekkäisiä prosesseja pääse syntymään ja sama prosessi voidaan käynnistää samalle oliolle uudelleen.

Prosessilla on *onInit*-metodin lisäksi *onPause* ja *onResume* -metodit, jotka määrittelevät mitä tehdään kun prosessi keskeytetään ja kun sitä jatketaan. Kuten pelimaailman olioilla, on myös prosesseilla sama alustusmetodin nimeämiskäytäntö. Prosessi alustetaan *initProcess*-metodilla, jossa jälkiosa nimestä koostuu itse olion nimestä. Syy on sama kuin pelimaailman oliolla. Itse prosessia päivitetään *onUpdate*-metodissa, jota prosessimanageri kutsuu pääsilmukassa. *Process*-oliossa metodi on käytännössä tyhjä, mutta johdetuissa prosessi-olioissa tämä metodi ylikirjoitetaan ja siinä tehdään suurin osa prossin hoitamasta työstä.

Process-oliosta johdetut oliot ovat lähes poikkeuksetta sidottuja enemmän peliin, kuin itse pelimoottoriin. Niissä määritellään mitä pelimaailmassa tapahtuu kun sitä päivitetään pääsilmukassa. Tämän työn rajoissa on määritelty viisi prosessia, jotka hoitavat pelimaailman tapahtumia.

GameProcess on pelimaailman tilaa päivittävä prosessi. Se hoitaa pelimaailman vierityskohdan päivittämisen ja pelattavan kentän siirtämisen sen mukaan. Tämän työn rajauksen sisällä ei tehdä erikseen fysiikkaprosessia, joten *GameProcess* hoitaa myös painovoiman vaikutuksen lisäämisen pelimaailmassa eläville oliolle, eli lähinnä pelaajalle. Se myös tarkastaa onko pelimaailman oliot vielä pelattavalla tasolla vai ovatko ne pudonneet sieltä pois.

Kun peli päättyy, *GameProcess* lopetetaan ja sen tilalle käynnistetään *EndGameProcess*, joka on sisällöltään hyvin lähellä *GameProcess*ia, mutta se hidastaa pelattavan tason vieritysnopeuden pysähdyksiin pienellä viiveellä ja kun tason vieritysnopeus on nollassa, se laukaisee tapahtuman joka näyttää pelin valikon.

Aikaisemmin mainittu *AnimatedActor* sisälsi animaation jolle annettiin tietty kuvataajuus. *AnimatedActorProcess*-prosessi on se joka hoitaa kuvataajuuden päivittämisen. Jokaiselle animoidulle oliolle annetaan tämä prosessi ja prosessin kohteeksi animoitu olio. Jokaisella päivityskierroksella kutsutaan tämän prosessin *onUpdate*-metodissa *AnimatedActor* olion *advanceFrame*-metodia, joka tarkastaa vaihdetaanko kuvasarjassa seuraavaan kuvaan.

Viimeinen työhön sisältyvä prosessi on *RollingActorProcess*. Tämä prosessi on sidottu pelimaailman taustakuvaolioihin, jotka ovat johdettuja *RollingActor*-oliosta. Tälle prosessille annetaan kohteeksi taustakuvaolio ja sen *onUpdate*-metodissa kutsutaan olion *setScrollPosition*-metodia, jossa päivitetään kuvan piirrettävää kohtaa.

Prosessien ajatuksena on antaa joustava ja helposti jatkettava, sekä modulaarinen lähestymistapa määritellä miten eri oliot käyttäytyvät pelimaailmassa. Melkein mikä tahansa tapahtuma mikä voidaan kuvitella tapahtuvan pelimaailmassa, voidaan eristää omaan prosessiinsa. Vaikka on välttämätöntä, että osa prosesseista on sidottuja tiettyihin peleihin, voidaan niitä myös tehdä osaksi pelimoottoria, jolloin saada yleisiä toimintoja suoraan osaksi sitä ja siten käytettäväksi monessa eri pelissä.

6.3.3 Pelattavat tasot

Jokaisessa pelissä täytyy olla alue, jossa peli tapahtuu sekä säännöt, jotka alueella pätevät. Pitää myös tietää mitkä oliot tällä alueella toimivat. Tätä varten on *Level*-olio. Sen tarkoituksena on toimia pohjana pelattaville tasoille. Sen tärkeimmät ominaisuudet liittyvät pelikentälle luotaviin *Block*-olioihin, jotka toimivat tasona, jonka päällä pelaaja voi kulkea. *Block*-oliot pitävät sisällään useita *ImageActor*-olioita, jotka muodostavat yhden tasopalan visuaalisen puolen. *Block*-olioita ei luoda käytetä suoraan, vaan niiden rakentamiseen käytetään BlockFactory funktiota (ohjelma 6). Näiden olioiden väliset suhteet näkyvät kuviossa 10.

Ohjelma 6 *BlockFactory* funktio ja *Block*–olioiden rakenne, joita sillä luodaan.

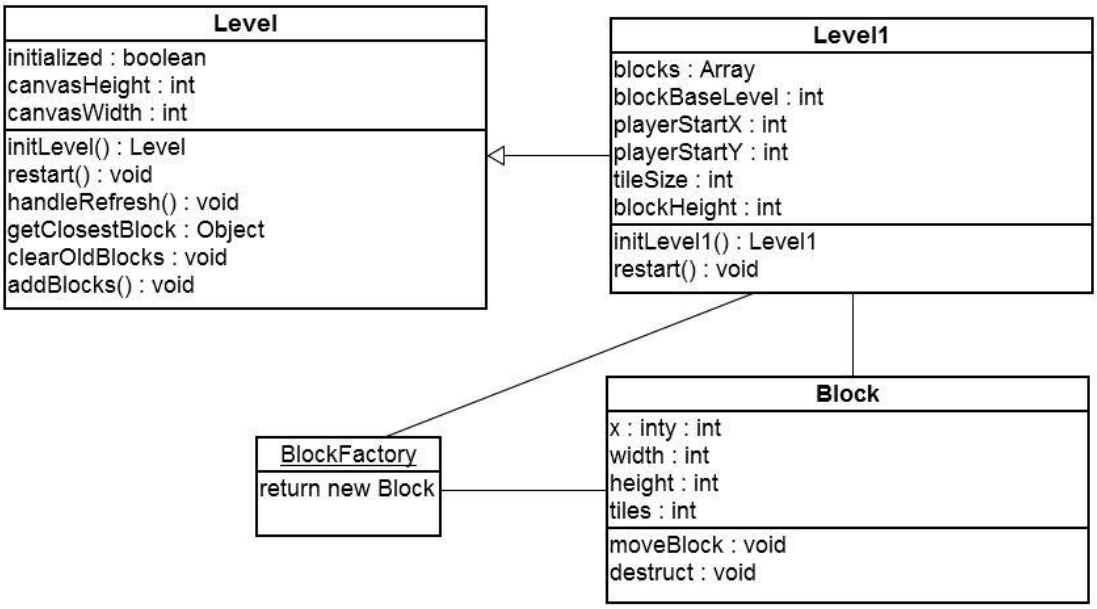
```
function BlockFactory(img, tileSize, height, x, y) {
    var width = tileSize * (Math.floor(Math.random()*6)+3);
    return new Block(img, x, y, width, height);;
}
```

Ohjelma 6 (jatkuu)

```
function Block(img, x, y, width, height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.tiles = [];

    for(var t = 0; t < this.width; t += 64) {
        var tile = new ImageActor();
        tile.initImageActor(this.x + t, this.y, 6, img);
        this.tiles.push(tile);
        g_Game.addActor(tile);
    }

    this.moveBlock = function(x,y){
        this.x += x;
        this.y += y;
        forEach(this.tiles, function(o){o.x += x; o.y += y});
    }
    this.destruct = function() {
        forEach(this.tiles, function(x) {x.destruct();});
        this.tiles.clear();
    }

    return this;
}
```

Level-olio sisältää kolme metodia, jotka huolehtivat tasojen luomisesta ja päivittämisestä. Sen *addBlocks*-metodi luo uuden taso-olion viimeisen pelimaailmassa olemassa olevan taso-olion perään jättäen satunnaisesti laskettavan välin

edeltävään olioon. Vanhat taso-oliot poistetaan *clearOldBlocks*-metodilla, joka tarkastaa, onko taso-olion loppupää mennyt jo pois ruudulta, ja jos on, se poistetaan kokonaan. Tasoa päivitetään kutsumalla *handleRefresh*-metodia, joka on käytännössä vain lyhyempi tapa kutsua kahta edellä mainittua metodia.



Kuvio 10 Pelattavien tasojen rakenne

Koska pelimoottorista on rajattu pois fysiikkamoottori ja sen prototyyppi toimii vain yhdellä tasolla, on yhteentörmäyksien tarkistuksessa oikaistu. *Level*-olion viimeisenä omana metodina on *getClosestBlock*. Se ottaa parametrina vastaan rajausalueen, jonka tarjoamista koordinaateista se etsii lähimmän taso-olion. Se palauttaa olion, jolla on kaksi attribuuttia. Palautettavan olion *object*-attribuutti viittaa lähimpään löytyvään taso-olioon, kun taas sen *type*-attribuutti kertoo onko parametrina annettu rajausalue horisontaalisesti sen sisällä vai ulkopuolella. Näillä tiedoilla *PlayerActor*, joka on prototyypissä ainoa joka käyttää tätä metodia hyväkseen, tarkistaa onko se vielä kentällä vai putoamassa aukkoon.

6.4 Pelimoottorin käynnistäminen

Kaikki ohjelmat tarvitsevat kohdan jossa ne käynnistetään ja tässä työssä rakennettu pelimoottori ei ole poikkeus. Itsessään sen sisältämät JavaScript tiedostot ovat hyödyttömiä. Ne tarvitsevat alustan, jolla toimia. Tähän tarkoitukseen käytetään HTML-sivua. Jotta työn rajoissa rakennettu pelimoottori toimii, tarvitsee www-sivulla olla Canvas-elementti, jolla on *id*-attribuutin arvona ”canvas”. Sen lisäksi täytyy sivulle hakea kaikki pelimoottorin tarvitsemat JavaScript-tiedostot. Jatkossa tämä toiminto kannattaisi eristää *GameApp*-kerrokseen. Itse peli käynnistetään *Main.js* –tiedostosta, jonka lyhyt sisältö näkyy ohjelma 7:stä.

Ohjelma 7 Pelimoottorin aloituskohta

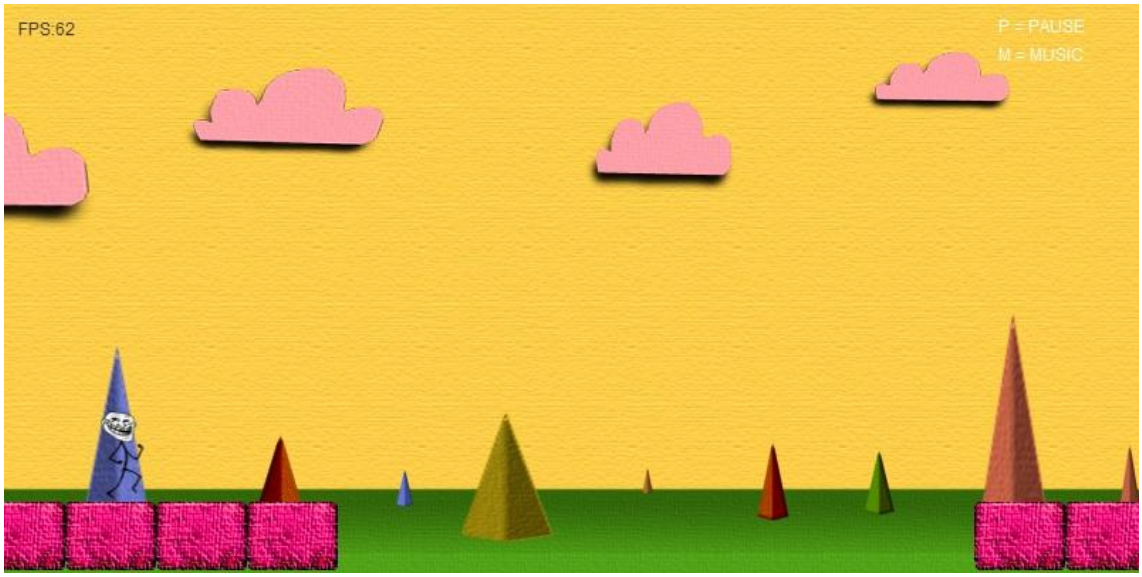
```
// Application entry point to be executed when document has loaded
window.onload = init;
/** Application entry point */
function init() {
    // Iitialize the GameApp and set a global reference to g_gameApp
    g_GameApp = new GameApp().initGameApp();

    // If the initialization went haywire, we don't run the app
    if(g_GameApp.init !== false)
        g_GameApp.run();// eternal loop
}
```

Kuten ohjelmasta 7 näkyy, on itse pelimoottorin käynnistäminen hyvin lyhyt operaatio. Kun sivusto on latautunut, niin kutsutaan *init* -funktiota joka luo uuden *GameApp* -olion. *GameApp* huolehtii pelimoottorin muiden osien alustamisesta. Kun pelimoottori on alustettu, katsotaan onko *GameApp*in *init*-attribuutti tosi. *Init*-attribuutti asetetaan todeksi ainoastaan, jos pelimoottorin alustus onnistui. Tällöin käynnistetään sen pääsilmukka *GameApp*in *run*-metodilla.

6.5 Prototyyppipeli

Pelimoottorin avulla pyörivä peli on todella yksinkertainen. Tämän työn tarkoituksena ei ollut rakentaa monipuolista ja tarkoituksellista peliä, vaan keskittyä alustan tutkimiseen ja pelimoottorin rungon rakentamiseen.

Kehitetty peli on sivulta päin kuvattu tasohyppely, jossa pelaajaa kuvastava olio yrittää pysyä kiihtyvällä vauhdilla eteenpäin kulkevalla tasolla putoamatta siinä oleviin aukkoihin. Kuvassa 1 on otettu kuvakaappaus pelistä toiminnassa.



Kuva 1 Kuvakaappaus pelistä toiminnassa

Pelissä on varsinaisesti vain kaksi tilaa. Valikko, jossa odotetaan että pelaaja aloittaa pelin ja itse pelattava osio missä pelaaja juoksee pelimaailmassa. Kun peli päättyy pelaajan pudotessa koloon, palataan takaisin valikkoon (kuva 2).

Kuva 2 Pelin valikko

Peli osaa ottaa vastaan kolme eri näppäinkomentoa. M-näppäin keskeyttää ja aloittaa taustamusiikin soittamisen, P-näppäin asettaa pelin tauolle ja välilyönnistä pelaaja hyppää. Valikossa oltaessa peli käynnistyy kun painaa mitä tahansa näppäintä.

6.6 Pelimoottorin ja pelin eroitus toisistaan

Koska pelimoottorin tarkoituksena on toimia alustana usealle pelille, on tärkeää että se saadaan hyvin erotettua itse pelistä. Tämän työn rajauksen sisällä tähän tavoitteeseen päästiin osittain.

Pelimoottorissa pyörivä peli koostuu pääosin vain muutamasta oliosta. Kappaleessa 6.3.1 mainituista peliolioista ainoastaan *PlayerActor* kuuluu pelille. Muut ovat yleismaailmallisia ja uudelleenkäytettäviä osia, joita voidaan käyttää hyväksi muissa peleissä. Prosesseista pelille on omistettu *GameProcess* ja *EndGameProcess*. Nämä kaksi prosessia ovat väkisinkin pelikohtaisia, koska ne vastaavat pelin olioiden ja sääntöjen päivittämisistä. Viimeisenä pelikohtaisena oliona on *Level1*-olio, joka kuvastaa pelattavaa maailmaa. Siitä tekee pelikoh-

taisen se, että se määrittää mitä resursseja ja olioita käytetään pelissä ja millainen pelattava taso on.

Kuten käy ilmi, itse peli koostuu vain neljästä oliosta, joista jokainen on johdettu pelimoottorin omista luokista. Tästä voi päätellä, että toiminnallisuuksien lisääminen ja uusien pelien tekeminen on huomattavasti kevyempää pelimoottorin arkkitehtuurin takia verrattuna siihen, että jokainen peli ohjelmoitaisiin alusta loppuun käyttäen käytännössä kertakäyttöistä koodia.

Työn rajauksen vuoksi pelimoottoriin on jouduttu lisäämään vain tähän peliin kuuluvia toiminnallisuuksia ja esimerkiksi eri tyylilajia edustava peli vaatisi pelimoottorilta lisää ominaisuuksia.

7 YHTEENVETO

Työn tavoitteena oli saada selville, soveltuuko HTML5 nykyisellään www-pelien alustaksi. Tässä suhteessa työssä kehitetty pelimoottorin prototyyppi onnistui todistamaan, että alustalla on paljon potentiaalia nousta seuraavaksi suosituksi www-tekniikaksi.

Työn aikana tuli kuitenkin selväksi, että HTML5 kehittäminen vaatii enemmän optimointia toimiakseen yhtä jouhevasti kuin vastaava Flash–sovellus ja että sen kanssa työskentely vaatii laajaa tietopohjaa.

Itse työn ohella valmistunut pelimoottori ei täyttänyt kaikkia odotuksia. Sen suorituskyky eri selaimien JavaScript–moottoreissa jätti toivomisen varaa, vaikkakin Google Chrome -selaimessa se suoriutui tehtävistään moitteetta. Osa syystä on tosin myös peliin ladattavissa resursseissa, jotka ovat huonosti optimoituja sekä raskaita. Koska pelimoottori osoittautui opinnäytetyön aiheena erittäin laajaksi, rajoitettiin sen arkkitehtuurin loppuun viemistä. Tämän takia myös itse sovelluksen koodia jäi pelimoottorin koodin sekaan. Tästä huolimatta arkkitehtuurin rakenne osoittautui oivalliseksi jatkokehityksen kannalta mahdollistaen eri toimintojen laajentamisen sulavasti.

Valmistunutta pelimoottorin prototyyppiä on tarkoitus jatkaa opinnäytetyön jälkeen valmiiksi kokonaisuudeksi, sillä prototyyppinä se oli onnistunut ja sen lopullinen arkkitehtuuri oli tarpeeksi lähellä alkuperäistä suunnitelmaa. Tämän takia valmistunutta tuotosta ei tarvitse hylätä ja aloittaa suunnittelua alusta.

Seuraavana askeleena on arkkitehtuurisuunnitelman saattaminen loppuun, pelimoottorin täydellinen erottaminen pelistä sekä fysiikkamoottorin suunnittelu. Tämän jälkeen moottorin käyttömahdollisuuksia kaupallisessa mielessä voidaan alkaa kartoittaa sekä suunnitella siinä toimivaa peliä.

LÄHTEET

[1] Adobe, “Flash to Focus on PC Browsing and Mobile Apps; Adobe to More Aggressively Contribute to HTML5”, [www-dokumentti]. Saatavilla: http://blogs.adobe.com/flashplatform/2011/11/flash-to-focus-on-pc-browsing-and-mobile-apps-adobe-to-more-aggressively-contribute-to-html5.html. (Luettu: 16.11.2011)

[2] Haverbeke Marijn, *Eloquent JavaScript*, USA: No Starch Press, 2011

[3] Djajadinata Ray, “Create Advanced Web Applications With Object-Oriented Techniques”, *MSDN Magazine*, 2007, [www-dokumentti]. Saatavilla: http://msdn.microsoft.com/en-us/magazine/cc163419.aspx (Luettu: 16.11.2011)

[4] Flanagan David, *JavaScript: The Definitive Guide*, USA: O'Reilly, 2001

[5] Crockford Douglas, *JavaScript: The Good Parts*, USA: O'Reilly, 2008

[6] Mozilla Developer Network, “Introduction to Object-Oriented JavaScript”, [www-dokumentti]. Saatavilla: https://developer.mozilla.org/en/Introduction_to_Object-Oriented_JavaScript. (Luettu: 16.11.2011)

[7] W3C, “Web Workers”, [www-dokumentti]. Saatavilla: http://dev.w3.org/html5/workers/. (Luettu: 25.11.2011)

[8] Fulton Steve & Fulton Jeff, *HTML 5 Canvas*, USA: O'Reilly, 2011

[9] Betabuild 2121, Sublime, http://www.sublimetext.com/2

[10] 1.7.6.msysgit.0, Git, http://git-scm.com/

[11] Wikipedia, “Agile software development”, [www-dokumentti]. Saatavilla: http://en.wikipedia.org/wiki/Agile_software_development. (Luettu: 16.11.2011)